1991-12

Time domain modal beamforming for a near vertical acoustic array

Crocker, Steven Edward

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TIME DOMAIN MODAL BEAMFORMING
FOR A NEAR VERTICAL ACOUSTIC ARRAY

by

Steven Edward Crocker

December, 1991

Thesis Advisor: James H. Miller

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) EC	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
8a NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

TIME DOMAIN MODAL BEAMFORMING FOR A NEAR VERTICAL ACOUSTIC ARRAY

12 PERSONAL AUTHOR(S)
CROCKER, Steven E.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1991 December	96

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US Government.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	acoustic tomography; beamforming; modal beamforming

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

Ocean acoustic tomography permits the mapping of various properties of a body of water through indirect means. The technique utilizes travel time variations for an acoustic signal to determine the structure of the ocean medium via inverse mathematical methods. The scale of any tomography experiment is fundamentally limited by the signal to noise ratio at the receiver. Through the use of a near vertical acoustic array, normal mode modeling of the local environment and a modal beamformer, array gains are possible which greatly extend the maximum separation between source and receiver. Additionally, the technique provides temporal resolution of the modal components of the arriving signal.

A time domain modal beamformer for a near vertical acoustic array has been developed. It has realized a nominal array gain of 6 dB for the Heard Island Experiment vertical array deployed off California. The primary obstacle to the technique remains inadequate array geometry description.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL MILLER, James H.	22b TELEPHONE (Include Area Code) 408-646-2384	22c OFFICE SYMBOL EC/Mr

DD Form 1473, JUN 86 *Previous editions are obsolete*
S/N 0102-LF-014-6603

Approved for public release; distribution is unlimited.

Time Domain Modal Beamforming for a Near Vertical Acoustic Array
by

Steven E. Crocker
Lieutenant, United States Navy
B.S., University of Lowell, 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING ACOUSTICS

from the

ABSTRACT

Ocean acoustic tomography permits the mapping of various properties of a body of water through indirect means. The technique utilizes travel time variations for an acoustic signal to determine the structure of the ocean medium via inverse mathematical methods. The scale of any tomography experiment is fundamentally limited by the signal to noise ratio at the receiver. Through the use of a near vertical acoustic array, normal mode modeling of the local environment and a modal beamformer, array gains are possible which greatly extend the maximum separation between source and receiver. Additionally, the technique provides temporal resolution of the modal components of the arriving signal.

A time domain modal beamformer for a near vertical acoustic array has been developed. It has realized a nominal array gain of 6 dB for the Heard Island Experiment vertical array deployed off California. The primary obstacle to the technique remains inadequate array geometry description.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This work would not have been possible without the support and encouragement of my family. It is they who deserve much of the credit for my success. My parents, Lincoln and Jacqueline Crocker, provided guidance when it was so desperately needed. My father taught me that with imagination and the will to succeed, nothing is impossible. I owe thanks to the Gillans for seeing in me, qualities that I didn't know existed. Most of all, I thank my wife Joyce and son Kevin for their undying love and support. They provided motivation when my own was lacking.

I. INTRODUCTION

A. THESIS SUMMARY

The objective of this thesis is to develop a software package to beamform acoustic signals used in ocean acoustic tomography. The goal of tomography signal processing is the precise measurement of acoustic travel time. Sound speed is a well understood function of temperature, pressure and salinity. Therefore, a fluctuation in acoustic travel time is indicative of changes in the environment through which the acoustic energy has passed. Once the travel time fluctuations are known, inverse mathematical techniques can be used to infer various properties of the ocean medium. [Ref. 1]

One source of travel time measurements is through the use of explosive devices. Although such signals are easy to generate, they may not provide the required precision owing to dispersion of the various frequency components in the impulsive signal. Additionally, such signals are difficult or impossible to replicate. [Ref. 2]

A better method that has been employed in recent years is the use of *maximal-length sequences*. The technique utilizes a pseudorandom, binary sequence as the phase modulation for an electronically generated bandpass signal. Maximal-length sequences are well suited to travel time measurements by

virtue of their deterministic nature, autocorrelation properties and simplicity.

The goal of the work described in the following pages is to provide sufficient gain to permit the recovery and decoding necessary for accurate travel time measurements for acoustic paths of extreme length (in this case 10,000 nautical miles). Specifically, the programming package should be able to:

- Exploit the spatial structure of the incident wave field to discriminate among signals arriving from different directions.
- Utilize information regarding time varying array tilt and depth to estimate the array geometry.
- Provide a stable *virtual array* by using the array geometry and modal structure of the immediate environment.
- Distinguish among the various modal components of the target signal.
- Provide both time domain and frequency domain analysis.
- Remain sufficiently flexible so as to permit the processing of arrays of differing construction in future tomography experiments.
- Provide some measure of fault tolerance with respect to the receiving array.

The remainder of this thesis is structured as follows:

- Chapter II. Acoustic Wave Propagation Theory
- Chapter III. Modal Beamforming
- Chapter IV. The Heard Island Array
- Chapter V. Results and Conclusions

Chapter II presents an introduction to acoustic wave propagation. The approach taken is that of normal mode propagation in a range independent channel.

The third chapter is an introduction to the concept of modal beamforming. It reviews frequency domain and time domain beamformers. The algorithm utilized in this study is derived.

Chapter IV describes the construction and subsequent deployment of the receiving array used in this experiment. Data acquisition and preprocessing of the acoustic signal are discussed.

The final chapter presents a description of the array dynamics encountered and frequency domain output from the software developed. Additionally, proposals for system integration and performance improvements are detailed.

Appendix A contains the source code which has served as the body of this work. In addition to the beamformer, various utility programs are included which provide for array geometry data reduction.

B. THE HEARD ISLAND EXPERIMENT

The performance of the software package is evaluated on a raw data set (vice synthetic data). The data was acquired during the Heard Island Experiment which occurred during the period January 23 - February 2, 1991. The purpose of the experiment was to determine the reliability of global acoustic

paths for tomographic analysis. Specifically, it is desired to establish the viability of these transmission paths for a proposed multi-national attempt to detect a decreasing trend in acoustic travel time. Such a change would indicate an overall warming along the path, providing the first convincing evidence to the existence of global warming [Ref. 1].

The signals were transmitted from the vicinity of Heard Island in the southern Indian Ocean. The site was selected because it is central to a web of open water paths extending through all the worlds oceans as shown in Figure 1.1.

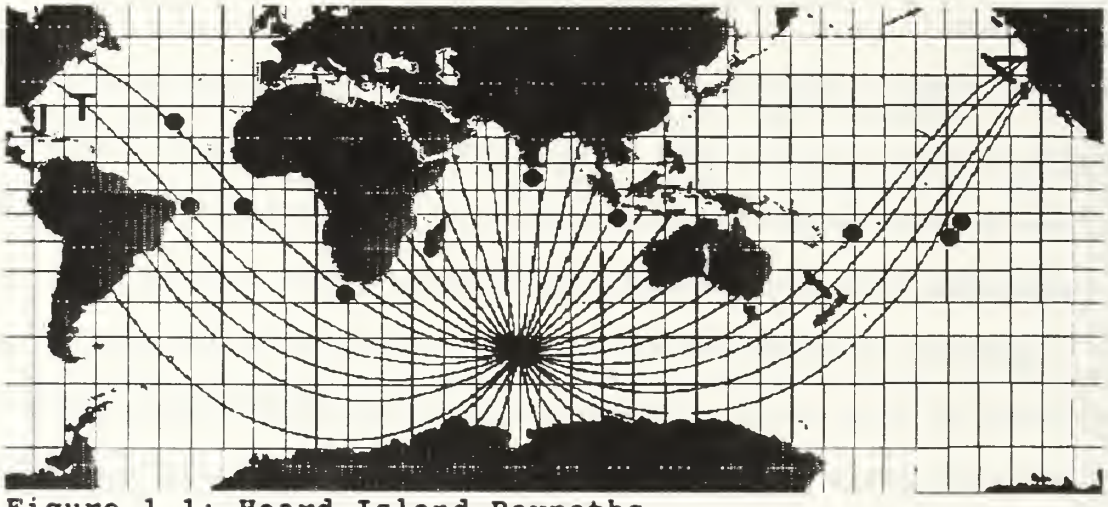

Figure 1.1: Heard Island Raypaths

The transmitting ship was the R/V Cory Chouest. The transmitter utilized a ten element vertical array with a maximum of five elements active at any time. A nominal source level of 209 dB re 1 μPa at 1 meter was realized. Individual projectors employed were HLF4LL very low frequency sources.

A variety of signal types were sent, including continuous wave and maximal-length sequences. A carrier frequency of 57 Hz was chosen both for its low absorption losses and the ability to distinguish it from the 50 and 60 Hz frequencies generated by power plants world wide.

The Monterey component of the Heard Island Experiment involved a collaboration between the Naval Postgraduate School, the Monterey Bay Aquarium Research Institute, the Massachusetts Institute of Technology, Woods Hole Oceanographic Institution, SAIC and the Moss Landing Marine Laboratory. The R/V Point Sur deployed a 32 element vertical array approximately 70 nautical miles south-west of Monterey Bay.

II. ACOUSTIC WAVE PROPAGATION THEORY

A. ACOUSTIC PROPAGATION IN AN OCEAN WAVEGUIDE

1. The Inhomogeneous Wave Equation

The *homogeneous linear wave equation* [Ref. 3],

$$\nabla^2 p = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2}, \qquad (2.1)$$

incorporates two implicit assumptions. The first restricts the sound speed to a constant value in the region of interest. Following the development of Coppens [Ref. 4], if the sound speed is now permitted to vary in space (but not time), then c=c(x,y,z) may be substituted into the wave equation without loss of generality.

The second implicit assumption restricts the use of the wave equation to sound fields which are free of sources. Before including a source term, the selection of a specific coordinate system is appropriate.

Given that sound in the ocean does not spread spherically in a free field, but radially with upper and lower boundaries, implies the use of a cylindrical coordinate system. To further simplify the problem to one sufficient for this work, the sound field shall be assumed to exhibit radial symmetry about the source, and sound speed variations shall be restricted to depth, c(z). Based on these restrictions,

application of the Lapacian yields the homogeneous, range independent wave equation,

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right)p+\frac{\partial^2}{\partial z^2}p=\frac{1}{c^2(z)}\frac{\partial^2}{\partial t^2}p. \qquad (2.2)$$

The inclusion of a source term requires that the inhomogeneous wave equation reduce to homogeneous form in regions which are free of sources of acoustic energy. This requirement implies that the delta function for a point source located at r=0 and $z=z_o$ have the form [Ref. 3]

$$\delta(\boldsymbol{r}-\boldsymbol{r}_o)=\frac{1}{2\pi r}\delta(r)\,\delta(z-z_o), \qquad (2.3)$$

where **r** is the spherical radius vector, r is the cylindrical radial coordinate and z is the cylindrical depth coordinate.

Inclusion of this term in the wave equation yields the Helmholtz equation for a monofrequency point source of unity amplitude,

$$\left[\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r}\right)+\frac{\partial^2}{\partial z^2}+\left(\frac{\omega}{c(z)}\right)^2\right]p=-\frac{2}{r}\delta(r)\,\delta(z-z_o)\,e^{j\omega t}. \qquad (2.4)$$

2. Solutions to the Inhomogeneous Wave Equation

Having adopted a cylindrical coordinate system, equation (2.4) can be solved by separation of variables. Furthermore, the complete solution can be treated as a linear

combination of normal modes,

$$p(r, z, t) = e^{j\omega t} \sum_m R_m(r)\, Z_m(z). \qquad (2.5)$$

The normal modes (Z_m) form an orthonormal set of eigenfunctions in z which satisfy the sourceless Helmholtz equation,

$$\frac{d^2 Z_m}{dz^2} + \left(\frac{\omega^2}{c^2(z)} - \kappa_m^2 \right) Z_m = 0, \qquad (2.6)$$

subject to the appropriate surface and bottom boundary conditions, and the orthonormal condition,

$$\int Z_n(z)\, Z_m(z)\, dz = \delta_{nm}, \qquad (2.7)$$

where κ_m is the eigenvalue for the m^{th} eigenfunction (or normal mode) and δ_{nm} is the *Kronecker delta function*.

A closed form expression for the $R_m(r)$ may be obtained by substitution of equation (2.6) into the inhomogeneous wave equation (2.4), multiplication of all terms by Z_n, integrating over z and application of the normalization condition (2.7). The result, after manipulation, is

$$\frac{1}{r} \frac{d}{dr} \left(r \frac{dR_m}{dr} \right) + \kappa_m^2 R_m = -\frac{2}{r} \delta(r)\, Z_m(z_o), \qquad (2.8)$$

the inhomogeneous Bessel's equation. It has the known solution

$$R_m(r) = -j\pi Z_m(z_o)\, H_o^{(2)}(\kappa_m r), \qquad (2.9)$$

where $H_o^{(2)}(\kappa_m r)$ is the *Hankel function of the second kind and order zero.* Substitution of this form into equation (2.5) yields

$$p(r, z, t) = -j\pi e^{j\omega t} \sum_m Z_m(z)\, Z_m(z_o)\, H_o^{(2)}(\kappa_m r). \qquad \textbf{(2.10)}$$

Closed form solutions for the $Z_m(z)$ are either difficult or impossible for all but select cases. Fortunately, efficient numerical algorithms exist which provide for the rapid solution of the depth dependent functions. [Ref. 5]

III. MODAL BEAMFORMING

A. PRELIMINARY CONCEPTS

Propagating waves can be modeled as functions of both space and time. Consequently, the attributes of such models can be used to extract information from real wave fields. Beamforming exploits the temporal and spatial characteristics of a specified environment to enhance a particular aspect of the wave field while attenuating undesirable components. Such an operation can be loosely termed constructive reenforcement of the desired signal or noise rejection. The information exploited in this development includes the modal structure of the immediate environment and the instrument data supplied by the array itself.

B. TIME DOMAIN BEAMFORMING

The acoustic signal received at a given hydrophone can be expressed as

$$p(t,x,y,z) = S(t,x,y,z) + N(t,x,y,z), \qquad (3.1)$$

where p is the pressure at the hydrophone, S is the desired signal and N is the local noise field. All are functions of time and hydrophone location.

If one now considers the signal received at an array of hydrophones, equation (3.1) takes the form

$$\boldsymbol{P}(t) = \boldsymbol{S}(t) + \boldsymbol{N}(t), \qquad (3.2)$$

where **p**, **S** and **N** are all N × 1 vectors representative of the individual array elements. The location of individual elements is implied by the vector index for a given hydrophone.

Classical (or plane wave) beamforming incorporates the use of a *complex steering vector* (**E**) to impose a phase shift on the individual elements in order to enhance the sensitivity of the array to signals propagating in a specific direction,

$$\boldsymbol{E} = \begin{bmatrix} e^{j\theta_1} \\ e^{j\theta_2} \\ \cdot \\ \cdot \\ \cdot \\ e^{j\theta_N} \end{bmatrix}. \qquad (3.3)$$

The output signal from such an array is

$$bf_{pw}(t) = \boldsymbol{E}^T(\boldsymbol{S}(t) + \boldsymbol{N}(t)). \qquad (3.4)$$

The subscript (pw) indicates that the beamformer assumes the presence of plane waves. Additionally, the implicit assumption is made in the above expression that the autocorrelation function of the noise field will be identically zero for any two (distinct) hydrophones. The success or failure of the beamformer will be a direct reflection of the validity of these assumptions.

If the desired signal is not monofrequency, but of finite bandwidth, simple phase shifting may not be suitable. This is particularly true if the target signal is carrying information such that the communication bit time is comparable to, or greater than the time taken for the wave to propagate across the array. Under such circumstances, the beamformer would impose phase shifts in distinct communication bit segments simultaneously. The resulting distortion to the bandpass signal is unacceptable in most communications applications. Beamforming under these conditions requires the application of true time delays vice phase shifts.

Assuming an array of spatial extent such that time domain beamforming is required implies a summation of the form

$$bf_{pw}(t) = \sum_{n=1}^{N} (S(t-\tau_n) + N(t-\tau_n)), \qquad (3.5)$$

where n is the index used to describe the array elements and τ_n represents the *steering delay* applied to the n^{th} element [Ref. 6]. This arrangement places no additional restrictions on the target signal characteristics or array construction in a given physical system.

C. THE MODAL BEAMFORMER

The signal model employed above is incomplete, in that it does not account for the nonuniform distribution of acoustic energy in the sound channel. Vertical arrays inherently

sample the modal structure of the immediate environment. Therefore, one needs to take the information provided by normal mode theory into account when the acoustic array spans a region of nonuniform energy density.

The appropriate refinement to the complex steering vector which incorporates the modal nature of the signal arrival is

$$\boldsymbol{E} = \begin{bmatrix} Z_1(z_1)\,e^{j\Psi_{11}} & Z_2(z_1)\,e^{j\Psi_{12}} & \cdots & Z_M(z_1)\,e^{j\Psi_{1M}} \\ Z_1(z_2)\,e^{j\Psi_{21}} & Z_2(z_2)\,e^{j\Psi_{22}} & \cdots & Z_M(z_2)\,e^{j\Psi_{2M}} \\ \vdots & \vdots & \cdots & \vdots \\ Z_1(z_N)\,e^{j\Psi_{N1}} & Z_2(z_N)\,e^{j\Psi_{N2}} & \cdots & Z_M(z_N)\,e^{j\Psi_{NM}} \end{bmatrix} \qquad (3.6)$$

where the m^{th} column of the matrix is the generalized steering vector for the m_{th} normal mode. The individual steering vectors now include amplitude factors ($Z_m(z_n)$) that reflect the value of the eigenfunction at the various hydrophone depths [Ref. 7]. The n^{th} row contains the values of those steering vectors at the depth of the n^{th} hydrophone. The phase terms (Ψ_{nm}) account for phase shifts required for the *beam steering* applicable to all modes (as required by array geometry) and phase delays among the various modes due to their individual propagation speeds. Specifically, the Ψ_{nm} take the form

$$\psi_{nm} = \theta_n + \alpha_m \ , \qquad (3.7)$$

where θ_n is determined by the hydrophone location and α_m is determined by the differences in propagation speeds over the transmission path for the normal modes present.

Modifying the steering matrix to implement time delays (vice phase shifts) is straight forward. A beamformer could be implemented by summing the output of individual hydrophones (**p**(t)) at this point. However, to do so implies a deterministic evaluation of the α_m. Such an evaluation would require near perfect knowledge of the conditions over the transmission path prior to commencement of the experiment. Given that the aim of tomography is the mapping of these conditions, such an approach is inappropriate.

An alternate approach is to utilize the columns of the matrix individually to discriminate among the modal components of the signal. Since all components of a given *modal steering vector* share a common value of α_m, these phase terms may be discarded from the individual elements of the steering vector. As a result of this simplification, the modal steering vector is completely determined by the Z_m (obtained from normal mode modeling of the immediate environment) and the desired *look direction* (determined by array geometry). Having established the methodology, and replacing the phase weights with steering delays, the output of the *modal beamformer* is

$$bf_m(t) = \sum_{n=1}^{N} Z_m(z_n)\, p_n(t-\tau_n)\,. \qquad \textbf{(3.8)}$$

where the subscript (m) indicates that the desired output is the component of signal representing the m^{th} normal mode. Expanding this form to explicitly include the terms

representing the signal and noise components of the acoustic field yields

$$bf_m(t) = \sum_{n=1}^{N} Z_m(z_n)\, S_n(t-\tau_n) + \sum_{n=1}^{N} Z_m(z_n)\, N_n(t-\tau_n) \qquad \textbf{(3.9)}$$

where S_n and N_n represent the amplitudes of signal and noise at the n^{th} hydrophone. The delay and weighting of the noise received at the hydrophones has no undesirable effects. If the noise between any two hydrophones was *uncorrelated* prior to the operation, then it will remain so after weighting. Again, the degree to which the ambient noise field is uncorrelated will ultimately reflect upon the performance of the beamformer.

IV. THE HEARD ISLAND VERTICAL ARRAY

A. ARRAY CONSTRUCTION

The array deployed for this experiment consisted of 32 hydrophones deployed vertically, each having a nominal sensitivity of -170 dB re 1 V/μPa. The element spacing was 45 meters, with hydrophone number one occupying a design depth of 345 meters. The nominal design depth for hydrophone number 32 was 1740 meters.

Instrumentation on the array consisted of two sensors. The upper sensor was located 4.0 meters above hydrophone number one. It recorded ambient pressure, temperature, tilt and current velocity. The lower sensor was located 5.0 meters below hydrophone number 20. It recorded tilt (including direction), pressure, temperature and conductivity. This sensor appears to have suffered a casualty during array deployment which rendered its tilt data suspect.

Flotation was provided by one main syntactic float and 28 syntactic *football floats*. The design called for the main float to reside at a nominal depth of 230 meters. Approximately half of the football floats were submerged. This arrangement rendered the array neutrally buoyant, thus providing a measure of isolation from surface wave effects. The array was directly tethered to the R/V Point Sur with

floatation devices on the surface portion. Each float was equipped with a flasher, a flag and a radar reflector. The float nearest the ship had the addition of a radio beacon. The procedure called for the ship to remain approximately 1200 meters from the position immediately above the array. During the experiment the R/V Point Sur was configured to remain quiet and dead in the water, except as required to keep clear of the array. Figure (3.1) illustrates the general configuration.

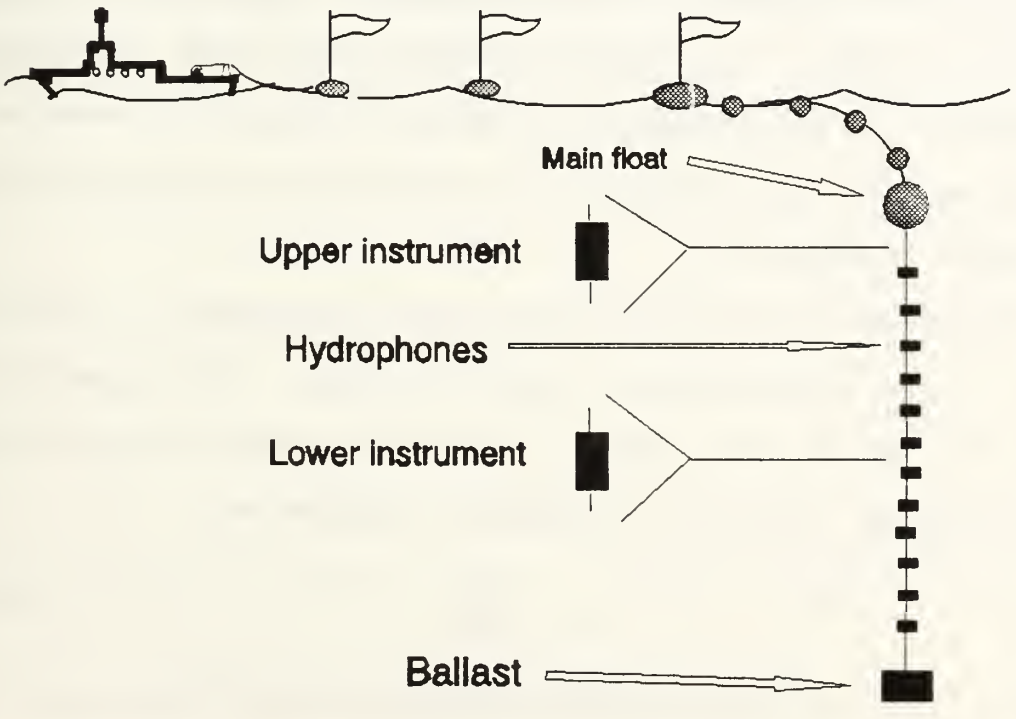

Figure 3.1: Heard Island Receiving Array

B. INSTRUMENT DATA

Ideally the receiving array, when deployed, is oriented vertically. However, due to currents, the requirement to maintain control of the research vessel and other factors beyond the control of the research team, the array is often tilted from the vertical. Given the length of the array, a 1.0° tilt results in a 24 meter horizontal displacement between hydrophones one and 32. This displacement is nearly one (carrier frequency) wavelength. Additionally, the direction in which the array tilts affects the component of horizontal displacement which is collinear to the propagation direction. Horizontal displacements perpendicular to the direction of propagation are acceptable if one assumes that the wave field is fairly uniform is this direction. Therefore, the *array steering* utilized in this work seeks only to correct for the component of displacement in the direction of signal propagation.

The speed with which the wave field propagates is required prior to the calculation of steering delays. The appropriate speed to use in this case is the *group speed* of the carrier frequency for the mode in question, defined as

$$c_{gm} = \frac{d\omega}{d\kappa_m} \qquad (3.10)$$

where c_{gm} is the local group speed for the m^{th} normal mode, ω is the radian frequency and κ_m is the appropriate eigenvalue.

The amplitude weight applied to a given hydrophone is a function of the mode number sought and hydrophone depth. The mode shapes ($Z_m(z)$) are required prior to beamforming. The numerical algorithm used to obtain the eigenvalues and eigenfunctions for this work was developed by Chiu and Ehret [Ref. 5]. It utilizes finite differencing of the propagation vectors for a given frequency and sound speed profile to calculate the modal structure of the local environment. The sound speed profiles were measured during the conduct of the experiment by the research team aboard the R/V Point Sur. Figure 3.2 is the sound speed profile used for this analysis. Figures 3.3 through 3.5 are the output of the normal mode model. Amplitude weights for the array elements were calculated from these data sets.

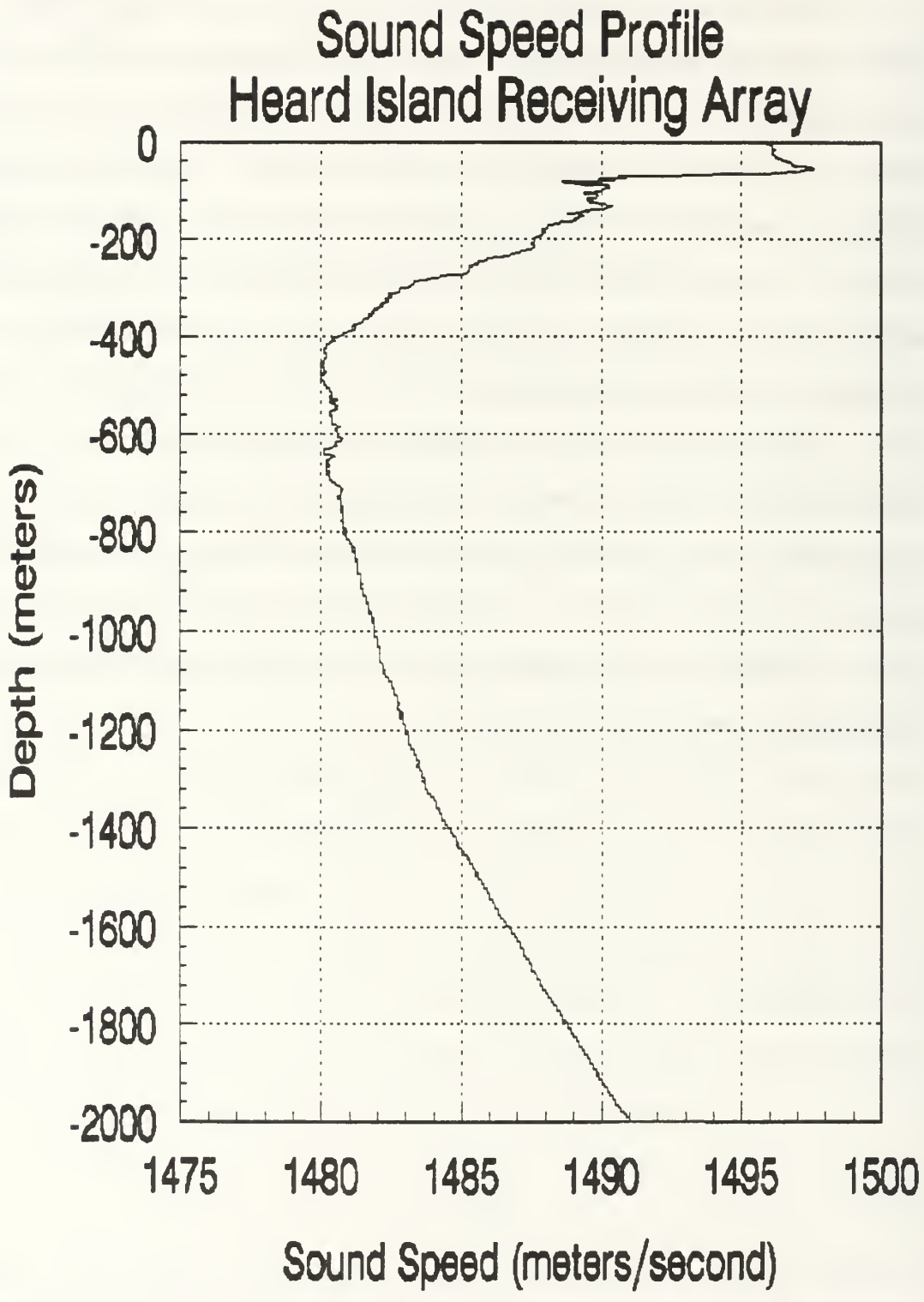

Figure 3.2: Sound speed profile at receiving array.

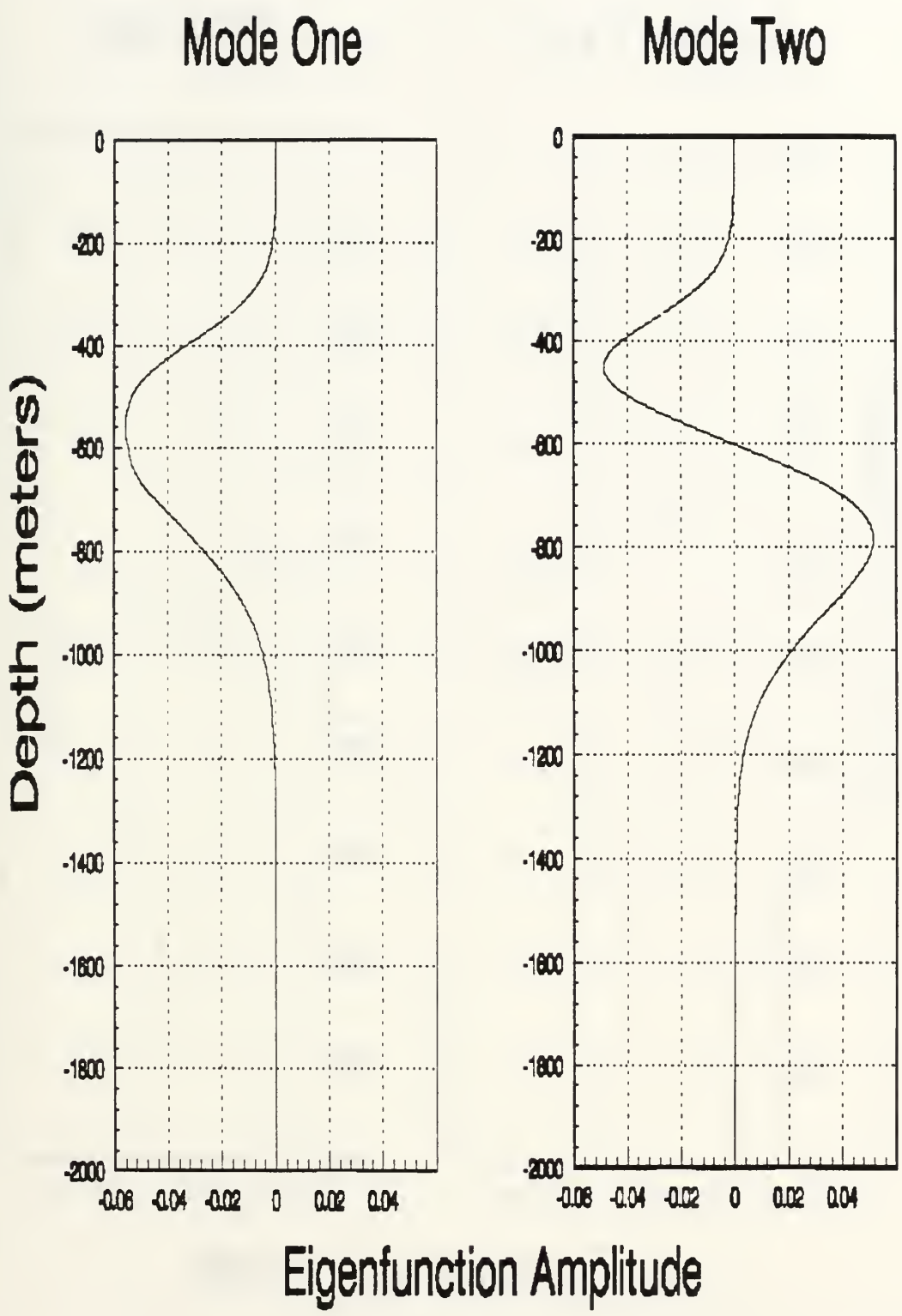

Figure 3.3: Normal modes one and two.

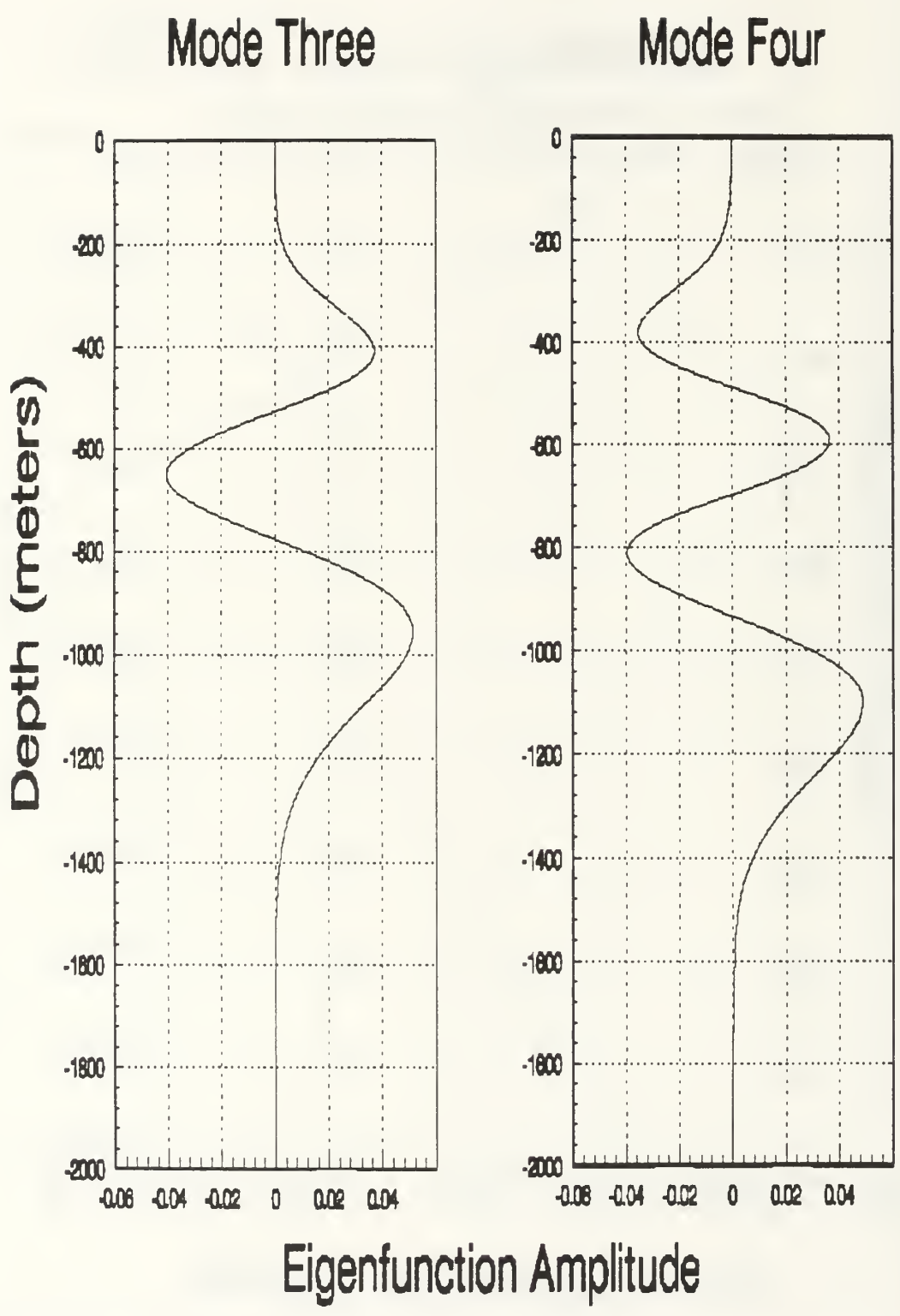

Figure 3.4: Normal modes three and four.

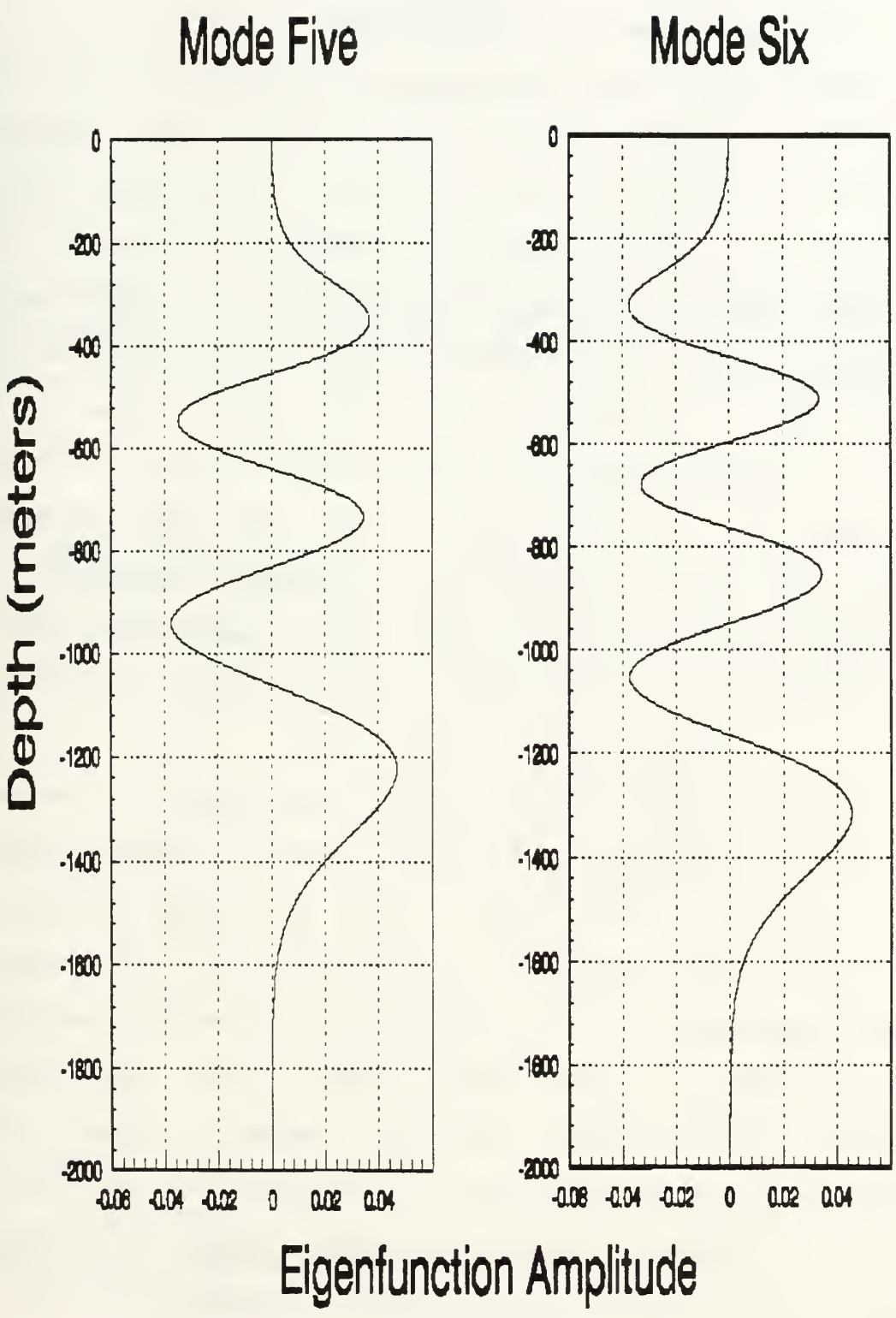

Figure 3.5: Normal modes five and six.

C. DATA ACQUISITION AND PREPROCESSING

Once deployed, the receiving array recorded numerous data sets, each lasting in excess of 60 minutes. Each recording comprises 32 channels of acoustic data. The hydrophone output was ported to a patch panel and preamplifier where a fixed gain of 26 dB was applied. Following this, the signal passed through a variable gain amplifier which automatically set the signal gain to optimize the 10 volt dynamic range of the analog to digital converter. This process had the duel effect of enhancing the precision of the data and preventing saturation of the A/D converters. Subsequent normalization of the data set to hydrophone output levels was accomplished by an implementation of a program written by Keith Von der Heydt of Woods Hole Oceanographic Institution.

Analog to digital conversion utilized a sampling frequency of 228 Hz. The signal was passed through an analog bandpass filter prior to conversion. This prevented aliasing of frequencies above *Nyquist* into the data set. Low frequency noise components were also attenuated. The cutoff frequencies for the bandpass filter were 10 and 80 Hz. Data storage utilized both optical disk and magnetic tape. Data acquisition and preprocessing of the data sets used in this work was conducted by Miller, Chiu, Frogner and others. Figure 3.6 illustrates the equipment alignment. [Ref. 8]

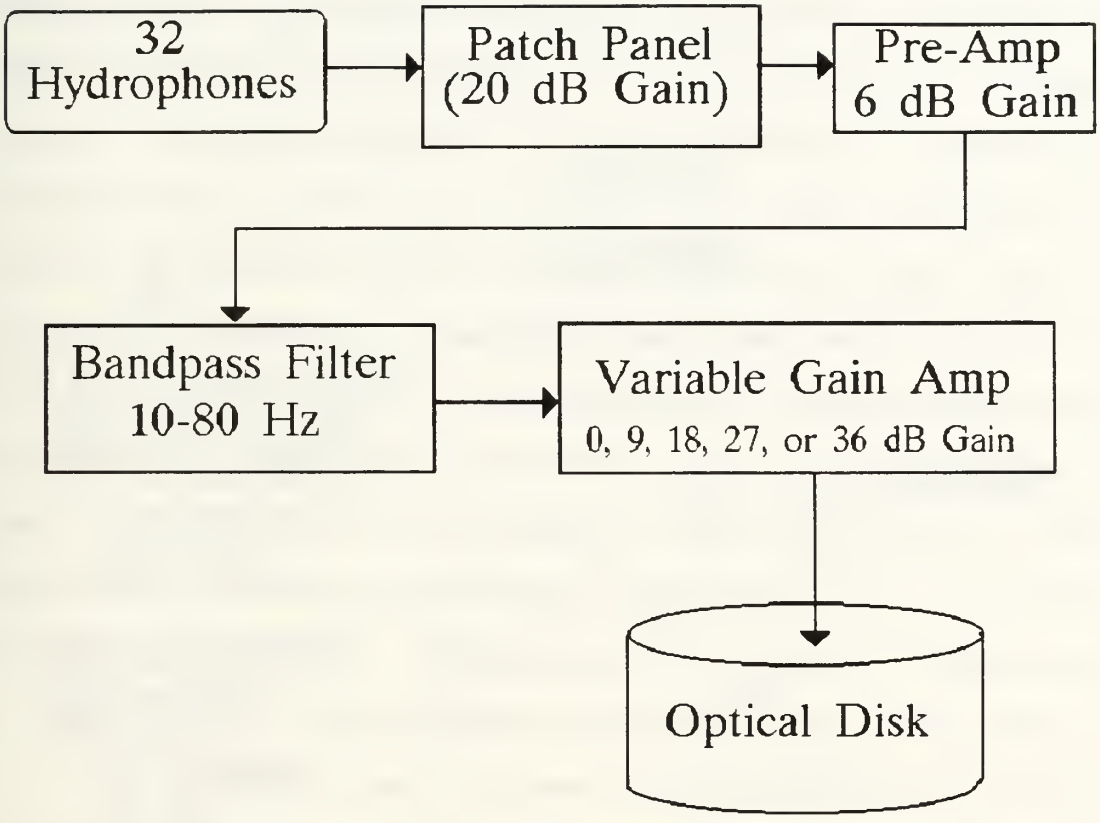

Figure 3.6: Data acquisition system

V. RESULTS AND CONCLUSIONS

The desired result of this research is the development of a software package capable of beamforming a tomography signal. The beamformer should discriminate among the various modal components of the signal, thus enabling one to treat the modes individually. Due to the inherent nature of vertical arrays, the software needs to incorporate information pertaining to the time varying array tilt and individual hydrophone depths. To this end, the beamformer utilizes a duty cycle. Each such cycle commences with an assessment of array geometry. This information is used to determine the appropriate steering delays and hydrophone weighting coefficients. Once established, the beamformer processes 60 seconds of acoustic data before repeating the procedure. In this manner, the software package implements a *quasi-stable virtual array* in which the *virtual aperture* remains fixed in space.

The results presented examine two performance aspects. The first pertains to array geometry description. Calculation of the steering delays and amplitude weights is detailed. Additionally, the acoustic output of the array is compared to that of a single hydrophone on the sound channel axis, thus allowing for a quantitative evaluation of the ultimate performance of the beamformer.

A. HYDROPHONE DELAY AND WEIGHTING

Information regarding array geometry was provided by two instrument packages. The upper sensor was located 4.0 meters above hydrophone number one, the lower sensor 5.0 meters below hydrophone number 20. Figure 5.1 illustrates the instrument configuration and coordinate system used in determining the array's spatial orientation.

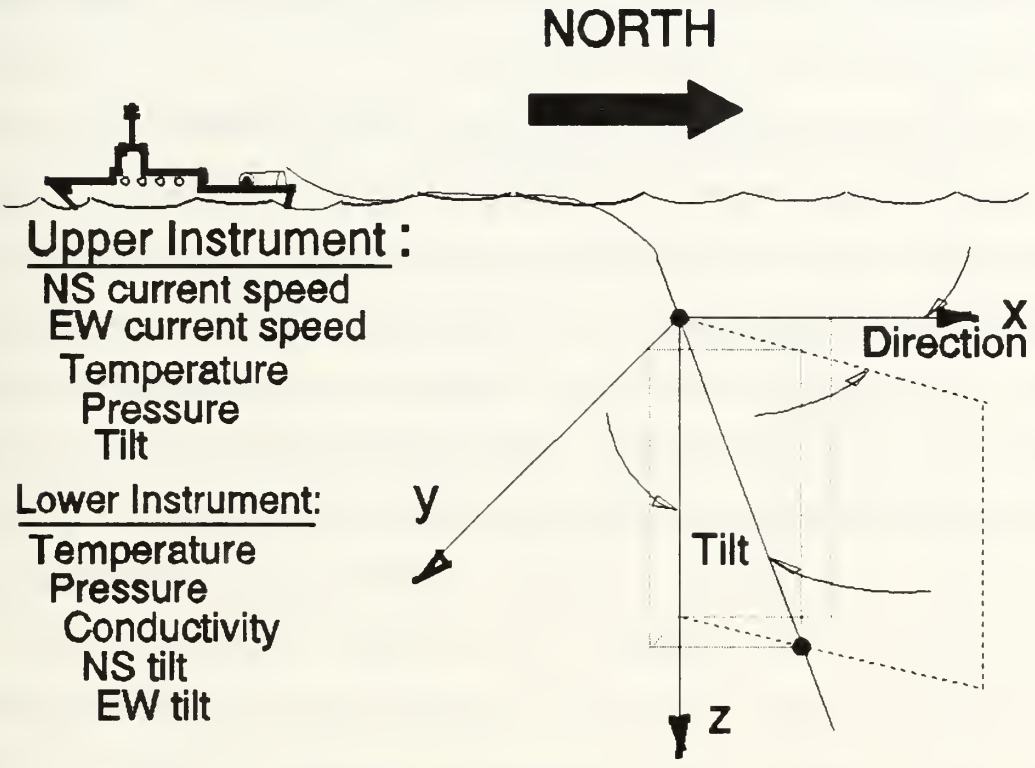

Figure 5.1: Array Instrumentation

1. The Upper Sensor

Information provided by the upper instrument includes the component of current velocity in both the north-south and east-west directions, ambient pressure, temperature and tilt. The direction in which the array tilts is determined by the current direction. The primary assumptions being that the current acting on the array is independent of depth and that the direction in which the array tilts is completely determined by the current direction.

In making these assumptions, one considers the primary source of current measured by the instrument to be a result of ship positioning adjustments. As mentioned previously, the R/V Point Sur was required to periodically move the ship in order to remain clear of the array. Such movements were intended to straighten the tether between the array and the towing rig, thus preventing the tether from becoming fouled in the ship's propellers. These adjustments resulted in a tensioning of the tether. It is this tensioning that is considered to have resulted in array movement and consequent water flow across the instrument package. Based on this assessment, the assumption that tilt direction was determined by the flow measured at the upper instrument is consistent with the conditions of the experiment. Higher order models could be derived for array dynamics, however this research has utilized the first order model of a linear array.

The next variable pertinent to array geometry is the pressure measured by the upper sensor. From this it is straight forward to calculate the sensor's depth using the following relationship,

$$z = \frac{P - 1\,atmosphere}{\rho g} \qquad (5.1)$$

where P is the pressure (in pascals) measured by the instrument, ρ is the density of seawater and g is the force of gravity.

Figures 5.2 and 5.3 present the data received from the upper instrument package during two of the acoustic recording periods. The recording beginning 00:05 (GMT) January 27 coincides with the reception of a m-sequence of length 2047. A continuous wave signal was received during the recording beginning 15:05 (GMT) January 27. They are representative of the most and least favorable conditions encountered during the course of the experiment.

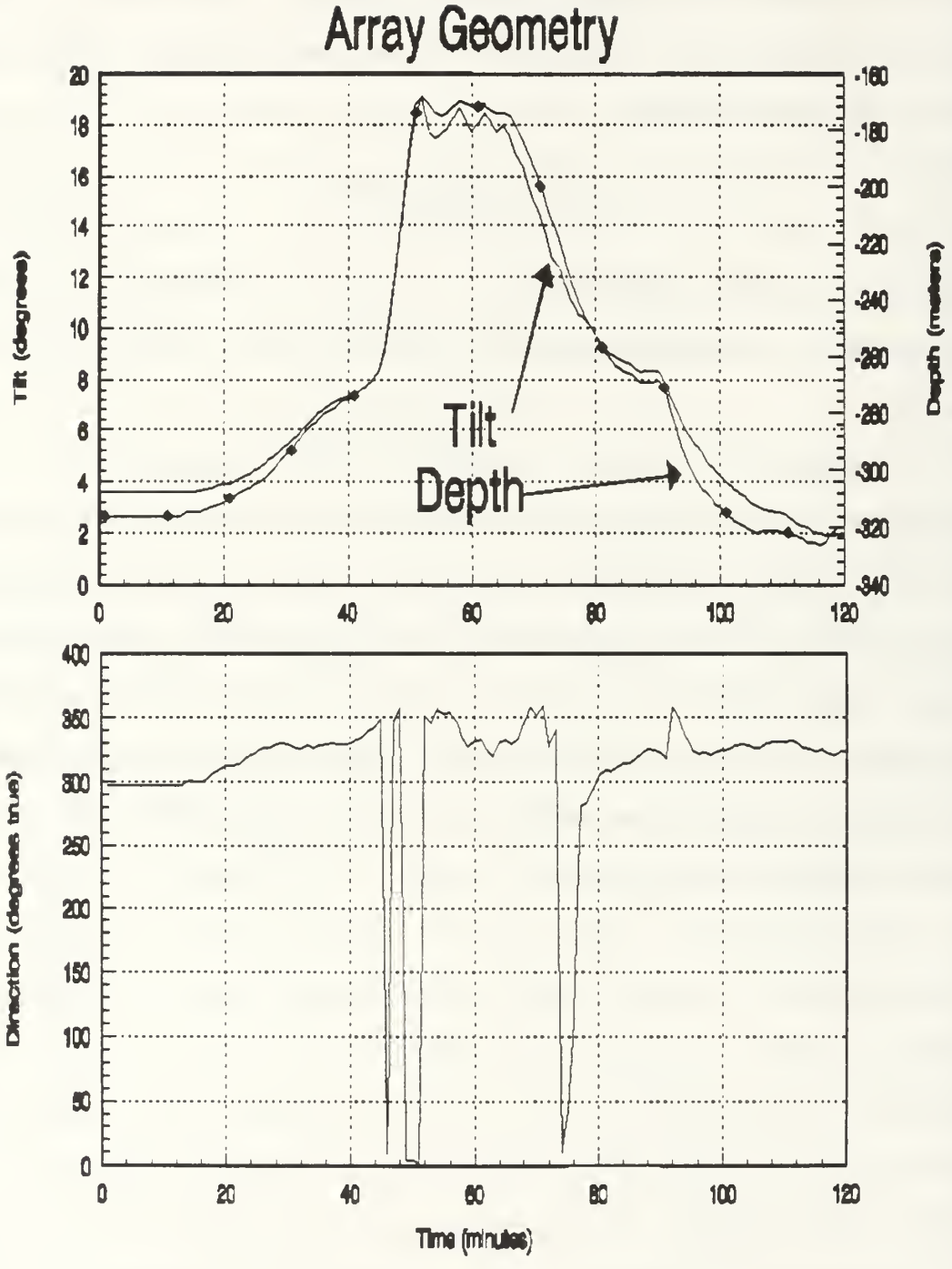

Figure 5.2: Array geometry for 00:05 (GMT) January 27.

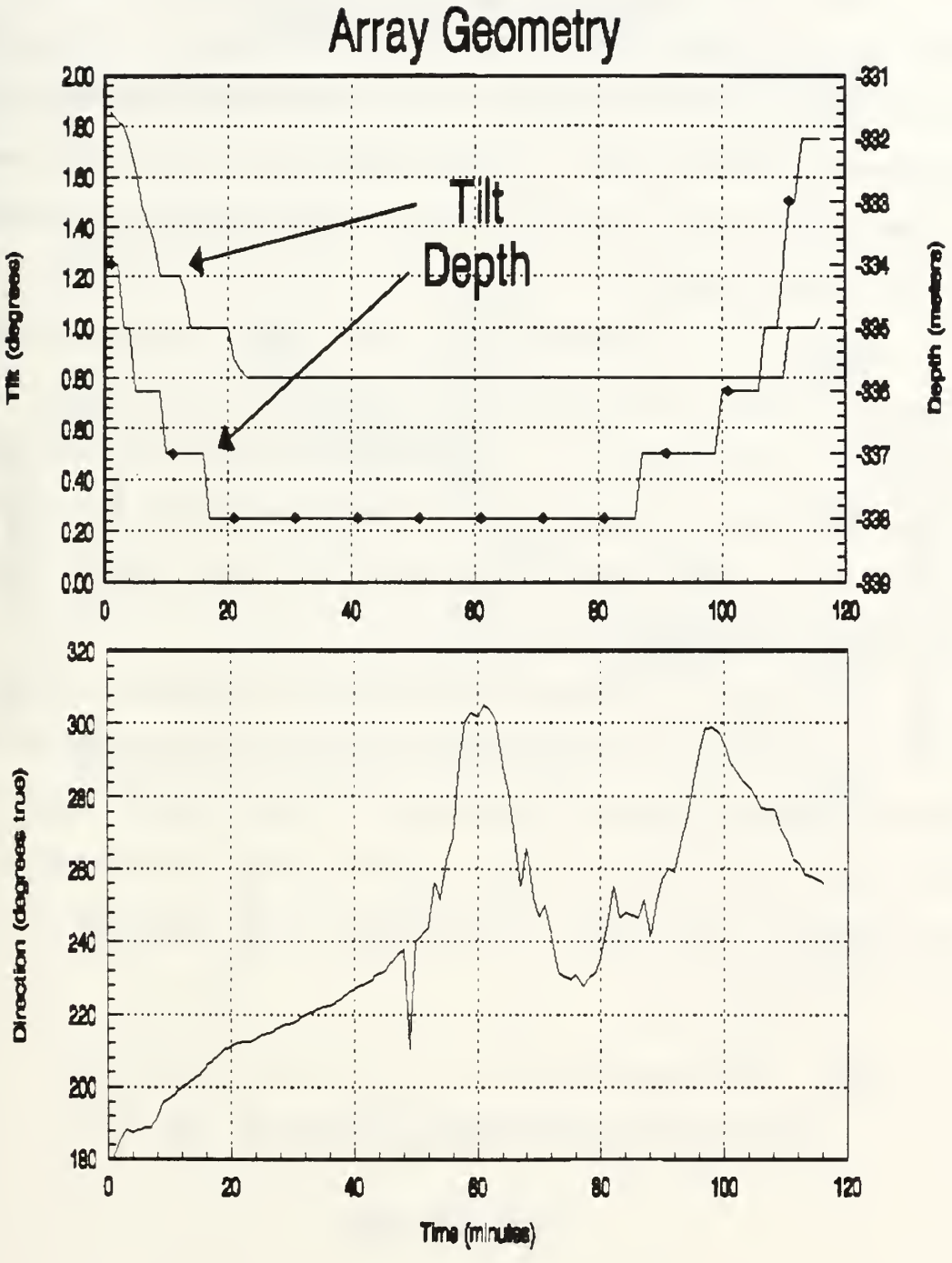

Figure 5.3: Array geometry 15:05 (GMT) January 27.

2. The Lower Sensor

The lower sensor was intended to provide a redundant source of array positioning data. Specifically, it measured tilt in the north-south and east-west directions, temperature, pressure and conductivity. As discussed previously, the lower instrument appears to have suffered a casualty which rendered the tilt data suspect.

Figure 5.4 indicates that an event occurred which caused the sensor to provide data which is not consistent with the assumed array geometry. One scenario explaining the event is that the sensor failed shortly after entering the water. Another possible explanation is that the array became entangled in itself upon descent.

All array tilt data was stored for subsequent analysis ashore. Therefore, the research team had no knowledge of the apparent failure during the conduct of the experiment. As there was no reason to suspect the instrument, no investigation was made to determine the source of the inconsistencies.

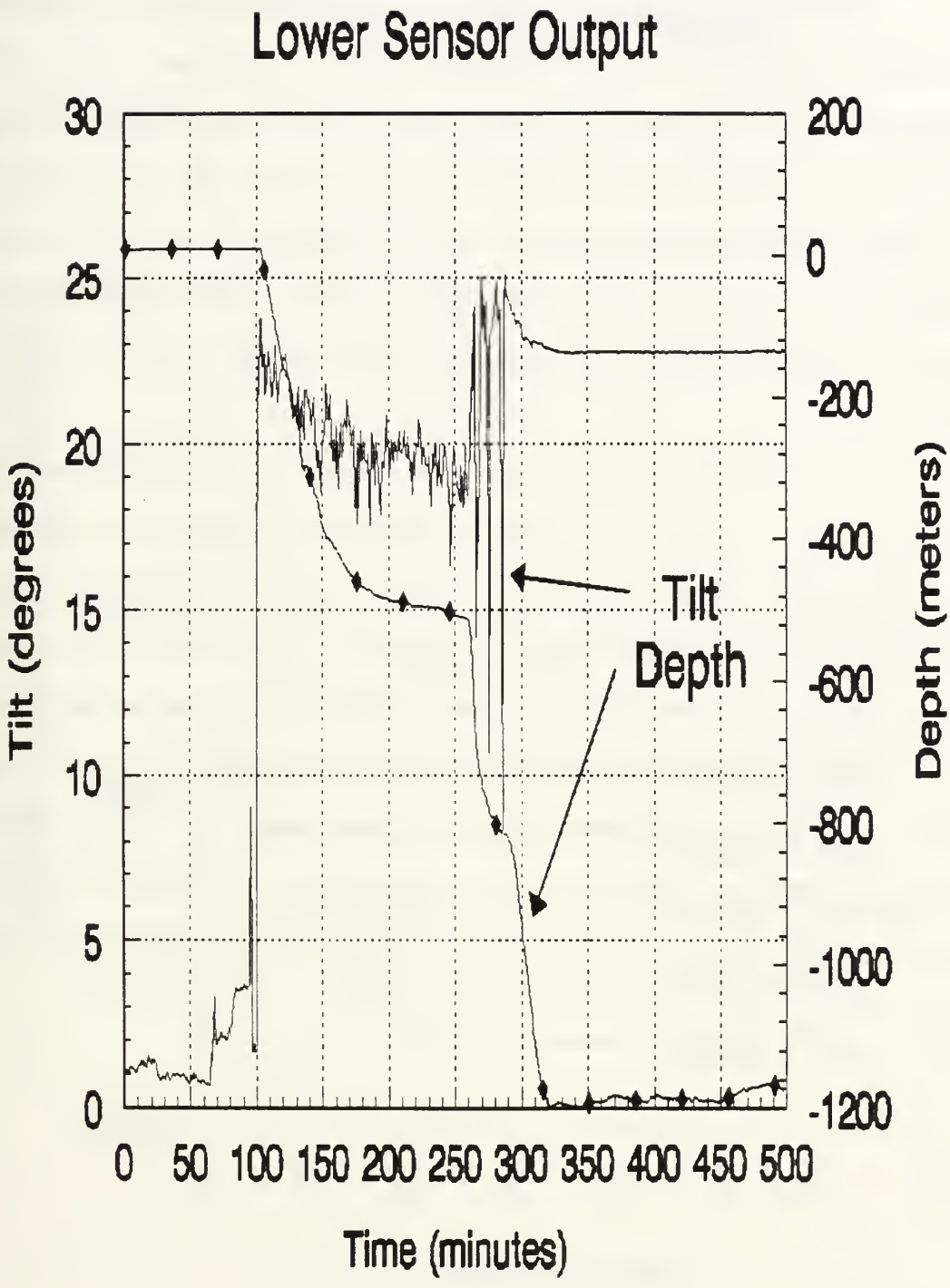

Figure 5.4: Lower sensor output upon deployment.

3. Steering Delays

Time delays must be applied to each of the hydrophones in order to correct for array tilt. However, only the component of horizontal displacement collinear to the signal propagation direction is used to calculate the time delay. This requirement mandates apriori knowledge of the propagation direction. Prior work has determined that the expected direction from which the signals would arrive is 217.0° true [Ref. 9].

In addition to array geometry and propagation direction, one requires a nominal propagation speed to convert the horizontal displacements to time delays. The appropriate speed to use is the *mode group speed*, as previously discussed. Group speeds for this study were obtained from eigenvalues calculated by the normal mode model courtesy of Chiu and Ehret [Ref. 5]. The mode group speed of the carrier frequency was used to calculate time delays. Figure 5.5 presents the mode group speeds as a function of frequency for the lowest five modes.

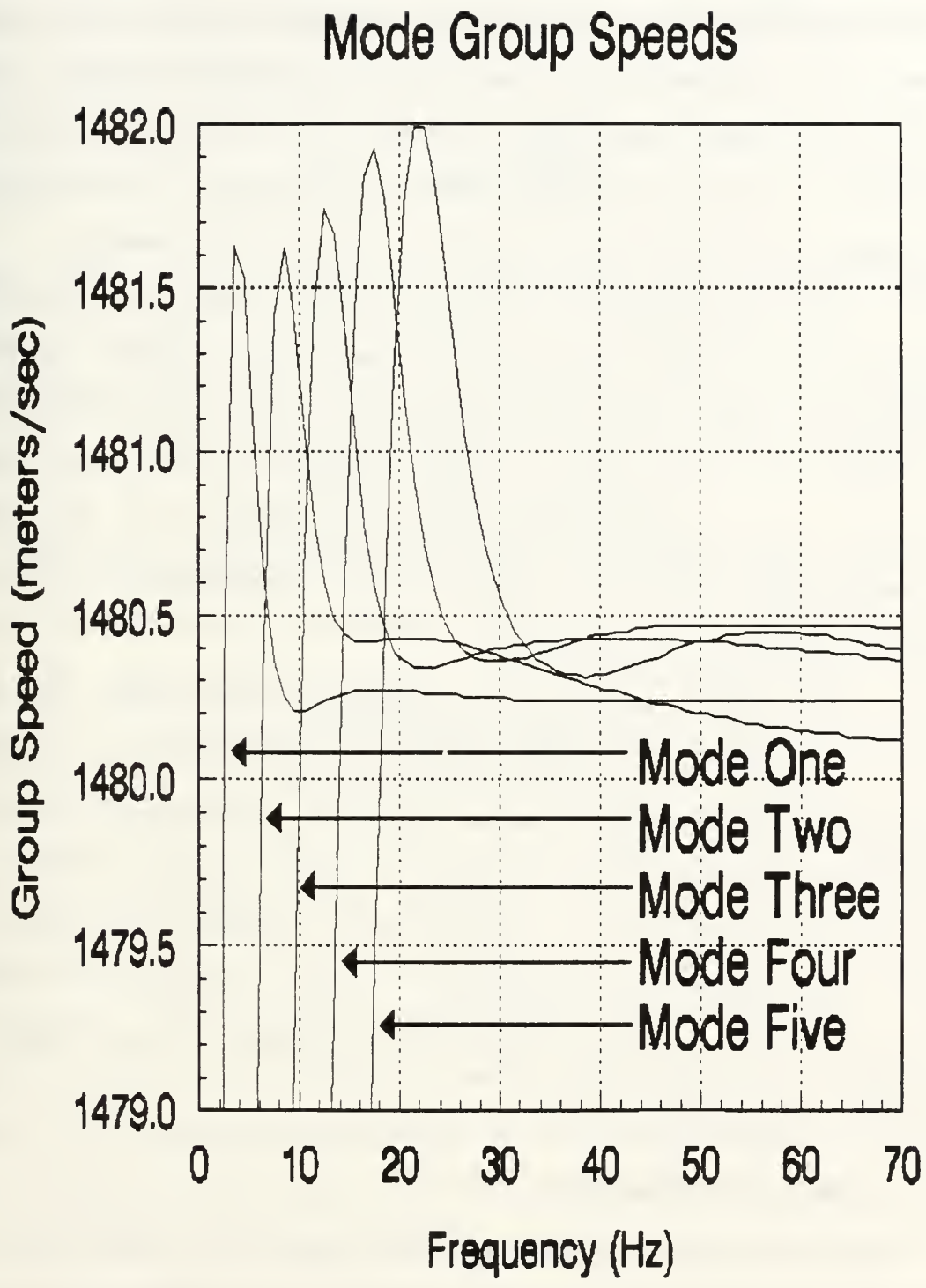

Figure 5.5: Mode group speeds.

4. Hydrophone Amplitude Weighting

Application of hydrophone weights utilizes the array geometry to determine the depth of each array element. The value of the depth dependent function (Z_m) is then assigned to the appropriate hydrophone output. This manner of weighting enhances the array sensitivity to the m^{th} normal mode.

An additional consideration encountered in hydrophone weighting is *fault tolerance*. Vertical acoustic arrays operate in a hostile environment, thus individual hydrophones may fail. The amplitude weights applied to the array must cope with the failure of individual elements. To this end, the beamformer applies a weight of zero to elements which have been evaluated as unreliable.

Log books kept by the research team aboard the R/V Point Sur indicating suspect hydrophones were verified by post processing statistical analysis of the individual elements. Several hydrophones were found to have failed during the course of the experiment. Specifically elements 18 through 28 were evaluated as unreliable. As a result of this determination, only hydrophones one through 17 were used to beamform the target signals.

B. ACOUSTIC PERFORMANCE

The preliminary data analysis seeks to quantify the beamformer's performance with respect to a single hydrophone located on the sound channel axis. The power spectrum

calculated from the output of hydrophone five was compared to the spectra calculated from the beamformed output of channels one through 17. The *array gain* is defined as

$$AG \equiv 10\log_{10}\left(\frac{SNR_A}{SNR_n}\right), \qquad \textbf{(5.2)}$$

where SNR_A is the signal to noise ratio of the beamformed output and SNR_n is the signal to noise ratio of the n^{th} hydrophone [Ref. 10]. The SNR for each case was calculated by integration of the power spectra over a 6.0 Hz bandwidth centered on the continuous wave frequency.

The array gain was calculated for numerous power spectra, each representing 60 seconds of acoustic data. Additionally, the gain was calculated for the lowest five normal modes present. Values for array gain range from a low of 2.5 dB to a high of 8.5 dB. The nominal array gain is 6 dB.

Reasons for the range in values include temporal variability in the modal components of the signal, errors introduced in the steering delays and errors in amplitude weighting. Time delay and amplitude weight errors are the direct result of assumptions made regarding array geometry. The primary source of geometric error is considered to involve the assumption that tilt direction is completely determined by current direction. Array tilt bearing may not be aligned well with current direction during periods when the direction of flow across the sensor is varying. Therefore, during periods

of variable current flow, marginal performance can be expected of the beamformer.

Figures 5.6 through 5.11 present representative power spectra of single channel data and beamformed data for modes one through five over the entire range of frequencies analyzed.

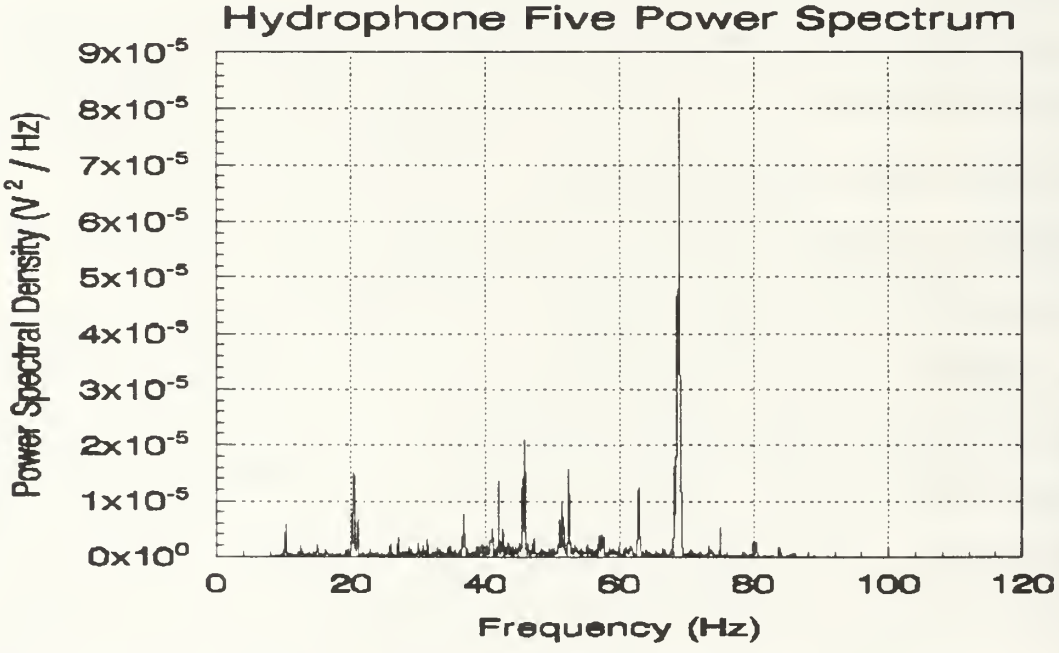

Figure 5.6: Hydrophone five power spectrum.

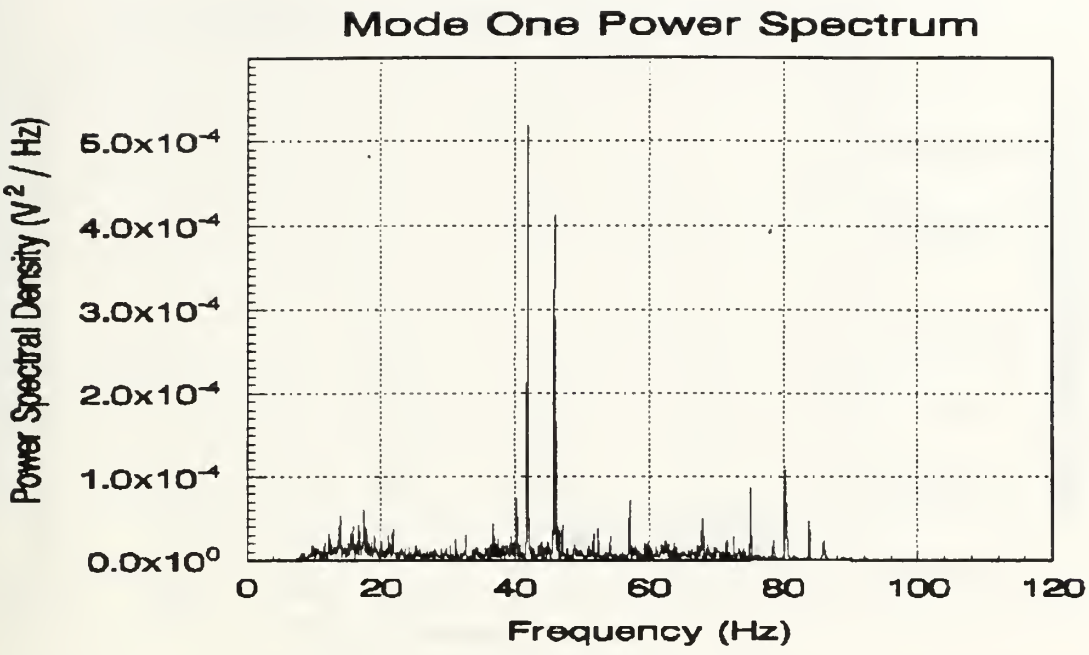

Figure 5.7: Mode one power spectrum.

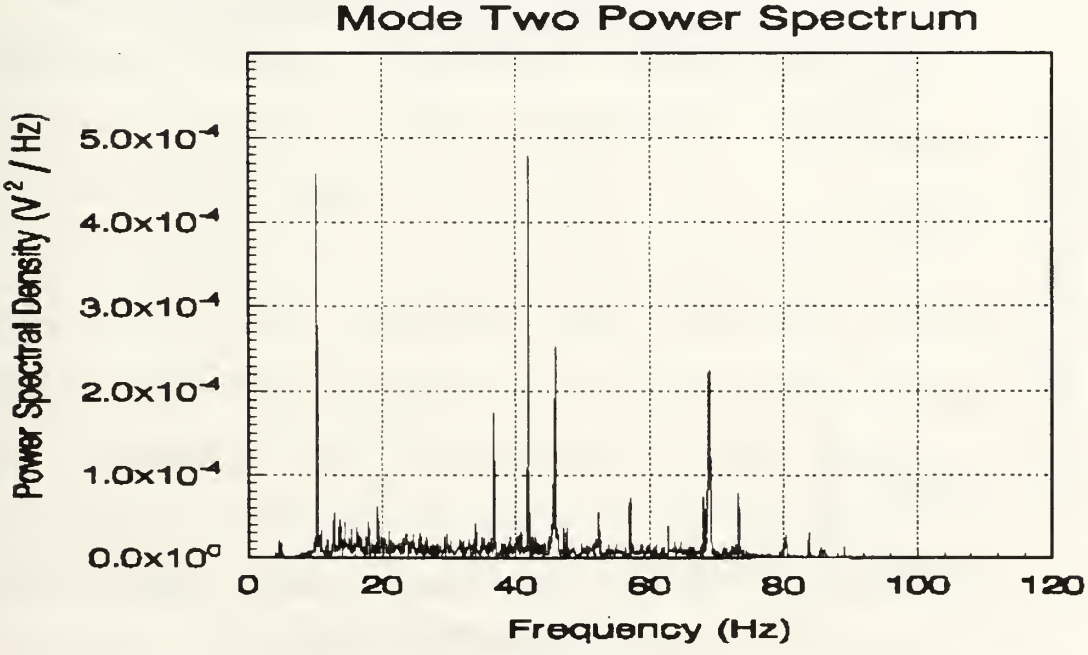

Figure 5.8: Mode two power spectrum.

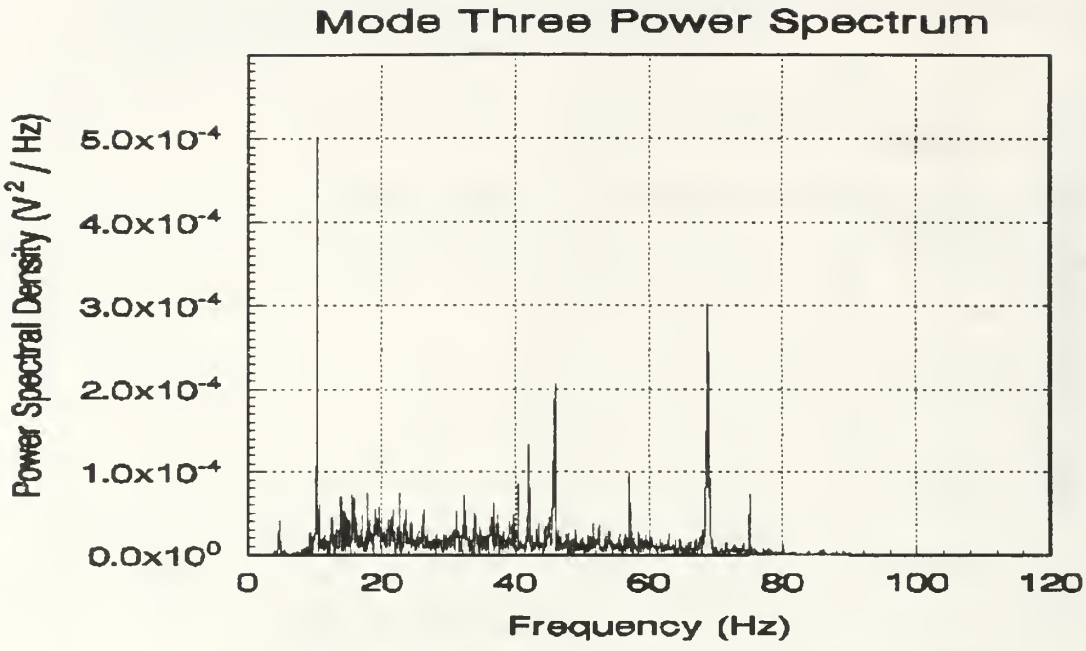

Figure 5.9: Mode three power spectrum.

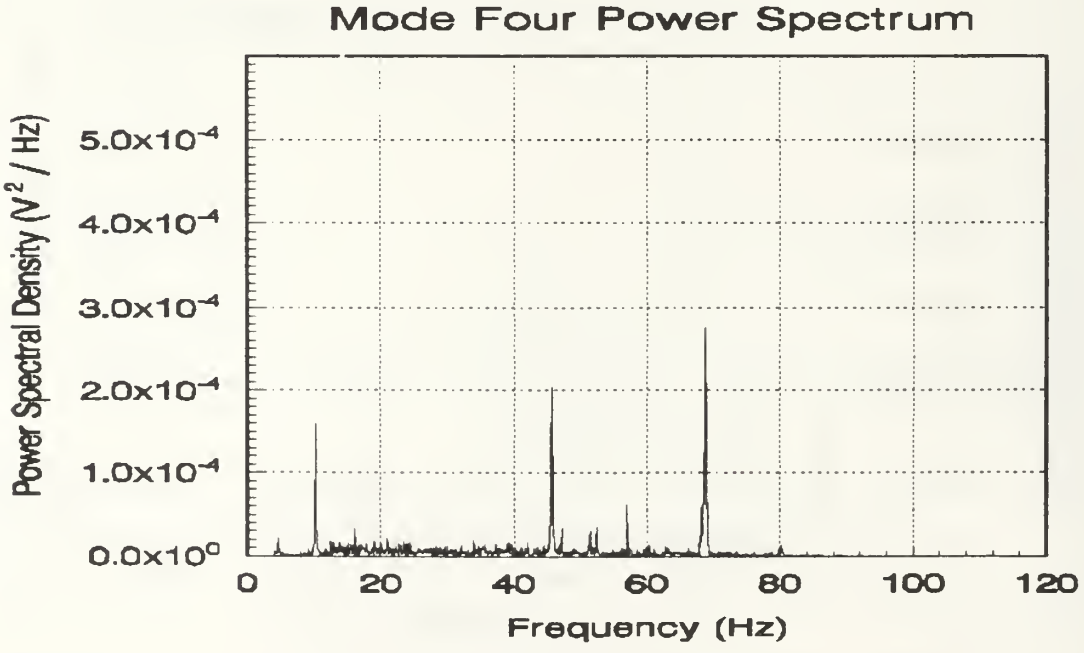

Figure 5.10: Mode four power spectrum.

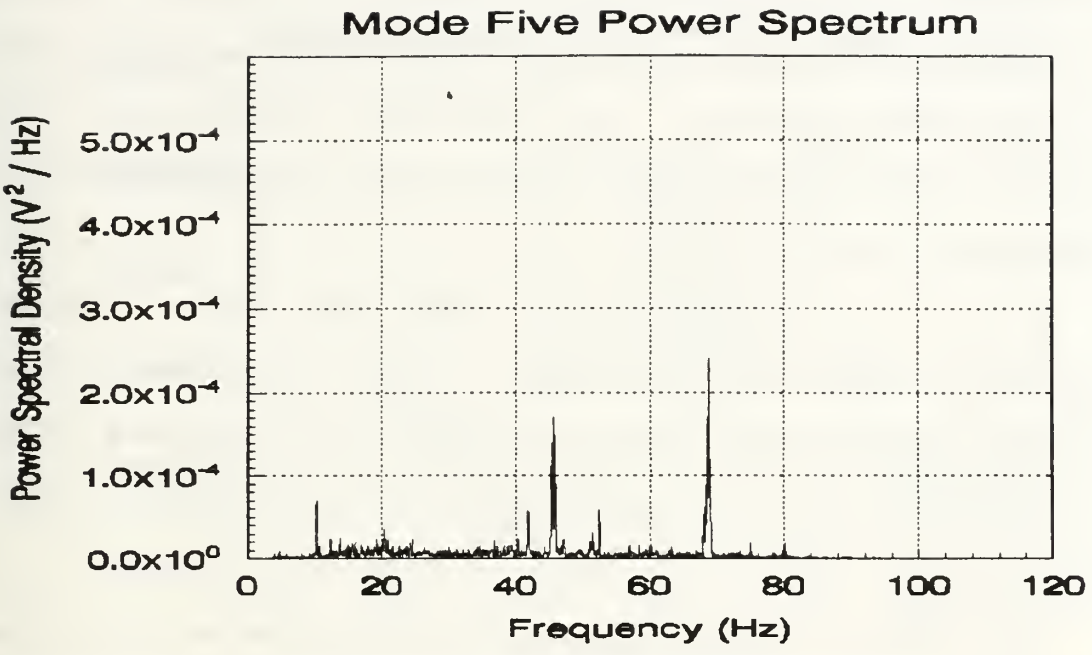

Figure 5.11: Mode five power spectrum.

As shown in the previous figures, the noise field at the beamformer is not incoherent. This fact, coupled with imperfect geometric array description, detracts from the calculated array gain.

One feature worth noting is the decreasing strength of the undesirable tonal components with increasing mode number, specifically those in the 40 to 50 Hz range. Presumably, these components were radiated by the R/V Point Sur. One hypothesis is that the strength of these signals in mode one is the result of *grating lobes* in the array's beam pattern for this mode. The amplitude weights applied to the hydrophones are similar to those for a plane wave beamformer, in that there is no phase shift applied along the array in the form of

coefficients of differing sign. Therefore, *spatial aliasing* is possible for frequencies with wavelengths less than twice the hydrophone spacing. For the Heard Island array, this includes frequencies ranging from 17 Hz to the bandpass cutoff frequency of 80 Hz.

As the mode number sought increases, these frequency components diminish in strength. This indicates that the rejection of coherent noise of local origin improves as the number of 180° phase changes applied to the hydrophone output increases.

Also shown in these figures is the presence of distant sources of coherent noise. These other tonal components are seen to vary both as a function of mode number and time. This observed behavior is consistent with modal propagation in a *range dependent* environment. Merchant traffic in shallow water and offshore drilling activity are two possible candidates for sources of coherent noise which becomes coupled into the sound channel.

Figures 5.12 through 5.17 present the same power spectra in the immediate vicinity of the carrier frequency (57 Hz). Again, the beamformed output indicates successful attenuation of undesirable components of a coherent noise field, presumably of local origin.

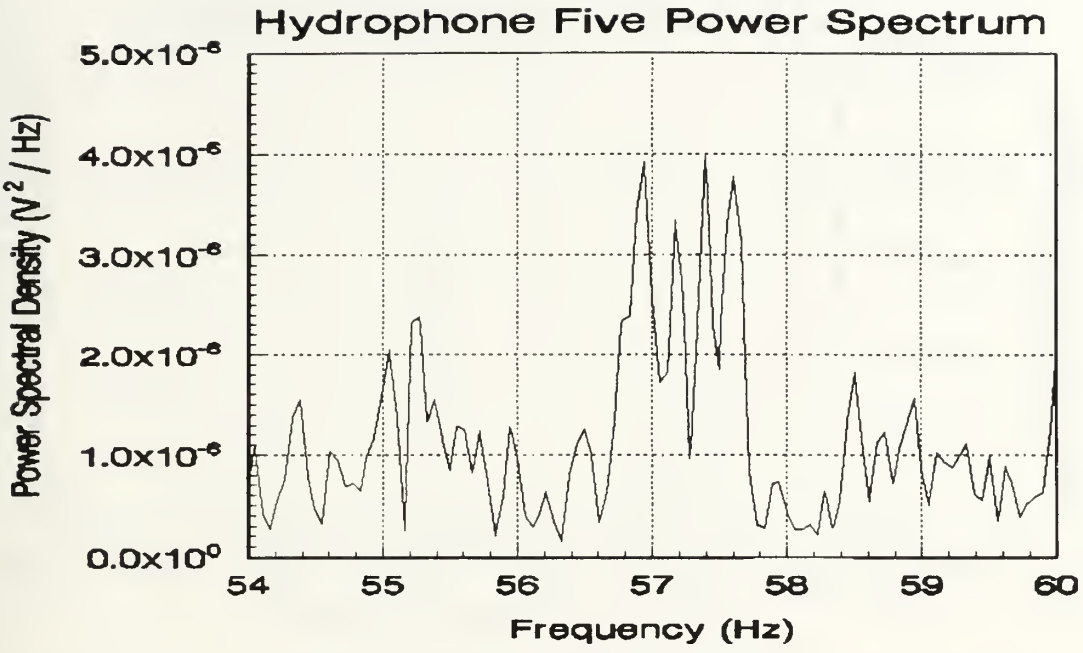

Figure 5.12: Hydrophone five power spectrum.

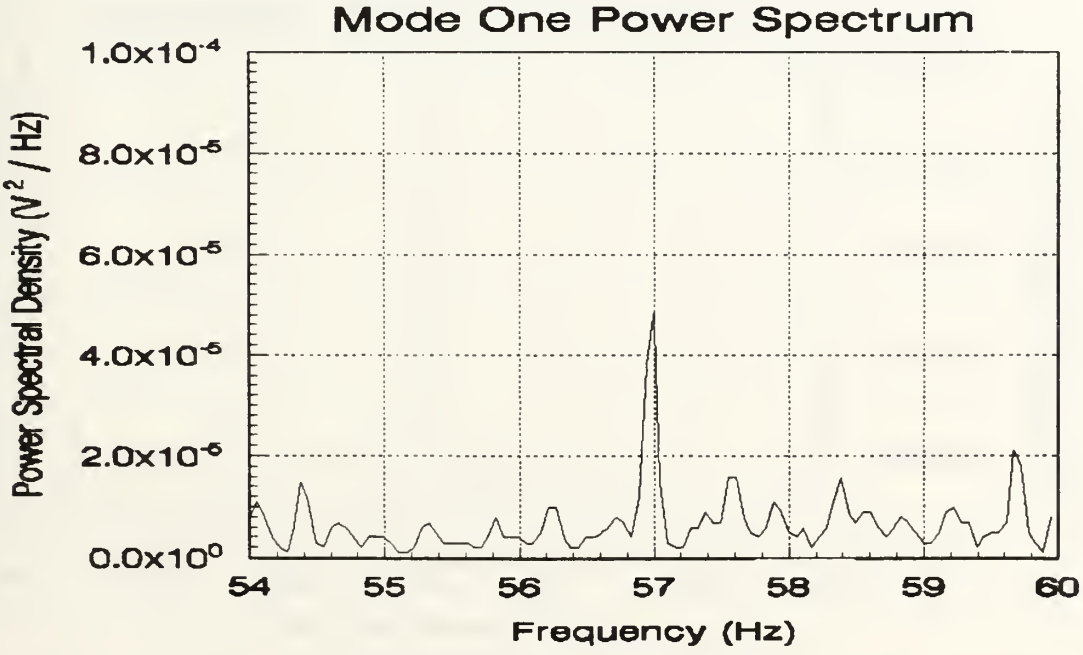

Figure 5.13: Mode one power spectrum.

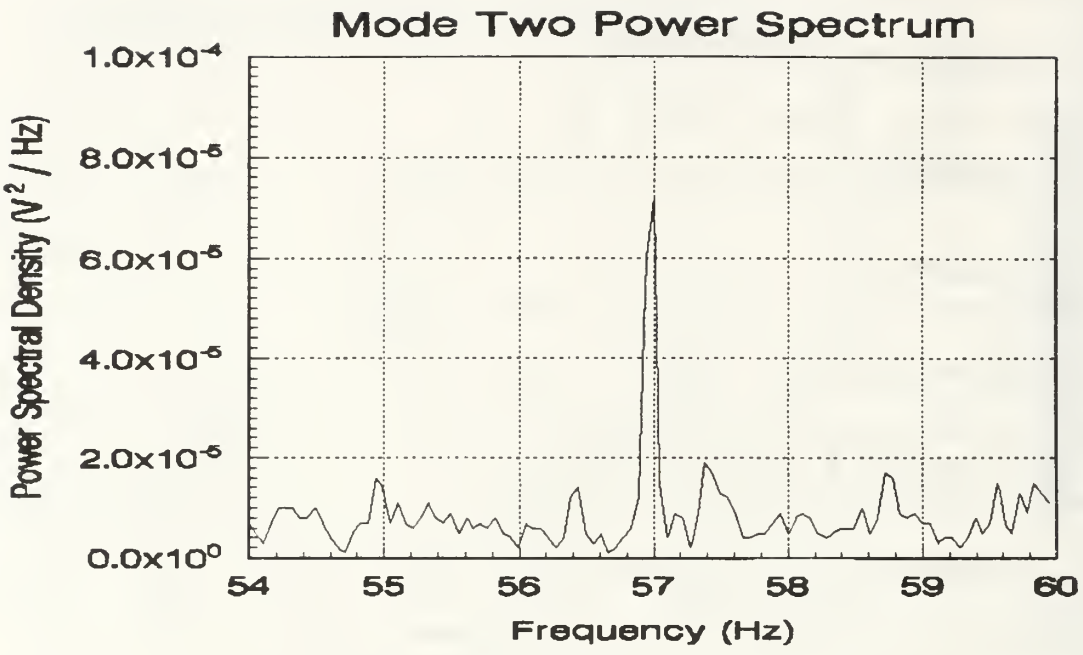

Figure 5.14: Mode two power spectrum.

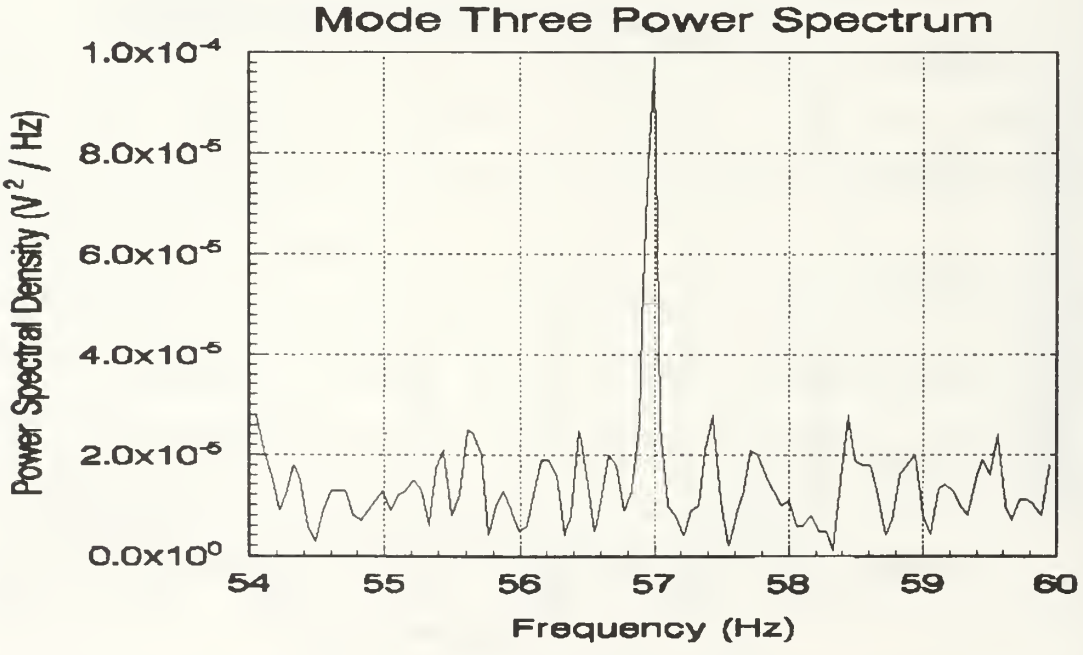

Figure 5.15: Mode three power spectrum.

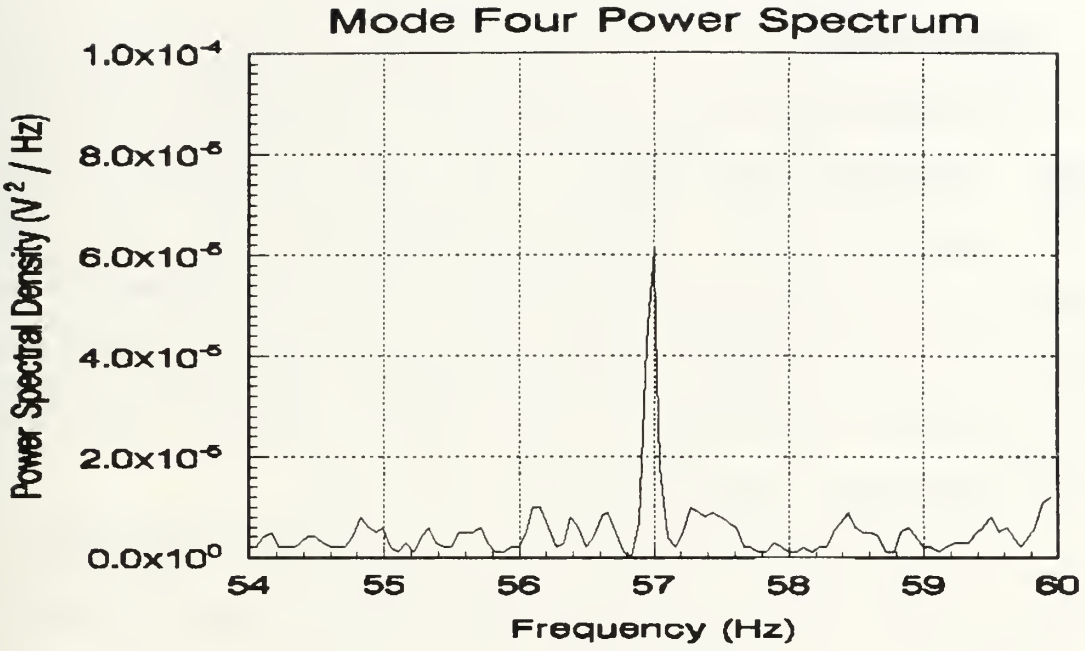

Figure 5.16: Mode four power spectrum.

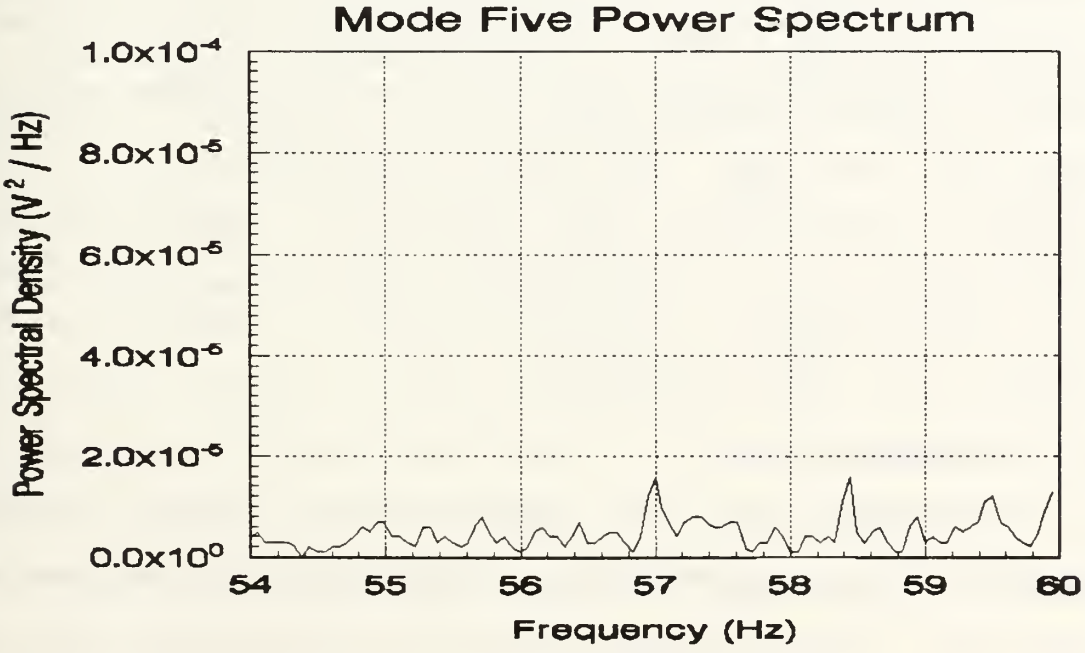

Figure 5.17: Mode five power spectrum.

C. CONCLUSIONS

Modal beamforming is a valuable technique for use in long range tomography experiments. The signals presented in this work traversed the longest transmission path for which reception was attempted during the course of the Heard Island Experiment. Although no travel time estimates have been made, the successful detection of the signals indicates that large scale tomography experiments are feasible.

The primary concern of this work was the development of a system capable of detecting the acoustic signals emanating from the vicinity of Heard Island. This has been accomplished. Additionally, the data describing the complex dynamics of the near vertical acoustic array has implications ıor the design and construction of vertical arrays for use in future tomography experiments. The most troublesome aspect of this work has been the accurate determination of array geometry, specifically tilt direction. Describing tilt direction in terms of measured current at one point on the array is considered inadequate.

D. RECOMMENDATIONS

Improvements to this implementation include physical aspects of the array itself and performance of the numerical algorithm. Large vertical arrays are particularly sensitive to errors in estimated geometry. In this experiment, an error

of 0.25° in tilt (assuming perfect knowledge of tilt direction) creates a phase error of 45 degrees between the most distant elements. This error is more destructive to the higher modes owing to their greater spatial extent. The problem is aggravated by questionable tilt direction data.

Construction of future vertical arrays should incorporate multiple instrument packages which directly measure tilt and direction. As this was the case with the lower instrument on the Heard Island array, it is most unfortunate that it failed. Additionally, an investigation into the utility of higher order models describing array shape would be illuminating. Incorporating a quadratic model for array shape would not impose a significant computational load on the beamformer and may improve the estimated position of individual hydrophones.

The current implementation of this beamformer utilizes third order polynomial interpolation during the application of time delays to individual hydrophone outputs. As such it represents the greatest computational load in the program. The interpolator is based on *Neville's algorithm* [Ref. 11]. No attempt was made to minimize execution time. As a result, the software requires approximately four hours to process one hour of acoustic data. A more efficient interpolator may reduce execution time sufficiently to permit this technique to

be used for real time modal beamforming. To this end, the software is assembled from modular components, thus allowing for improvements on a function by function basis.

APPENDIX A

The following programs were developed during the course of this research. They include the modal beamformer and associated utility programs. These programs may be obtained by contacting James H. Miller at his address in the initial distribution statement.

A. THE MODAL BEAMFORMER

1. Operational Considerations

The modal beamformer was designed with portability a prime consideration. As a result, there are numerous program parameters which must be set by the end user. Among these are the characteristics of the physical array and data acquisition system. All such user selectable parameters are contained solely within the program definitions. The parameters appear below in the same order in which they appear in the program.

- UNIX VERSION: selectively compiles either unix or ansi compatible code blocks.
- ASCII: selects output in either ascii or binary format. Ascii is useful for data export, binary saves storage space.
- SIGNAL: permits user to enable or disable the beamformer's time series output.
- SPECTRUM: permits the user to enable or disable the beamformer's power spectrum output.

- LOWER_SENSOR: directs the program to load or ignore lower tilt sensor data. This option should remain OFF as there is no reliable lower sensor data.

- VALIDATE: enables program validation. If selected the program will create output files which list the array geometry, steering delays, hydrophone weights and calculated mode group speed. This feature is useful for program debugging.

- ERROR_ESTIMATE: if selected, this feature causes the polynomial interpolator to store the upper bound on the mean squared error in the steering delays. This feature doubles the output of the beamformer.

- ON: logical switch for program control.

- OFF: logical switch for program control.

- INTERPOLATE: selects interpolator. This feature permits the installation of an improved interpolator without the requirement to search for and replace each call to the function.

- ORDER: indicates the degree of the polynomial used in the polynomial interpolator.

- STEP: indicates the number of points on either side of a sequence to be taken when estimating derivatives.

- TINY: prevents division by zero in floating point operations.

- PI: used for trigonometric recursions.

- RADIAN: used for degree to radian conversions.

- OFFSET: indicates the distance (in meters) between the number one hydrophone and the upper tilt sensor.

- DELTA_R: indicates array element spacing (in meters).

- CTD_OFFSET: indicates the difference in depth between the sound speed profile's first data point and the depth increment used in the profile (in meters).

- SSP_LENGTH: indicates the maximum number of data points that an eigenfunction may contain. The only restriction applicable to this parameter is available memory.

- EIGVAL_LENGTH: indicates the maximum number of eigenvalues that the program will accept. The only restriction applicable to this parameter is available memory.

- LOOK_DIRECTION: the compass direction (in degrees true) from which the signal arrives.

- TILT_BUFFER: the buffer length allocated to tilt data. The only restriction applicable to this parameter is available memory.

- BUFFER_TIME: the length of time (in seconds) represented by the acoustic data input buffer. This period represents the output buffer length plus one second on either end to permit steering delays and/or advances. The array is currently capable of *end firing* a 1500 meter acoustic array. This parameter must be an integer. Additionally, 60 must be an integer multiple of (BUFFER_TIME-2). The array has a duty cycle of one minute between design changes. There must be an integer number of output buffers per duty cycle.

- F_SAMPLE: sampling rate of the data acquisition system. This value must be an integer.

- CHANNELS: the number of hydrophones on the array.

- F_CARRIER: the carrier frequency of the target signal. The group speed used in calculating steering delays is based on this frequency. This must be entered in floating point.

- FFT_LENGTH: the length of the FFT used if frequency domain output is selected. This value must be a power of two which is less than or equal to the number of samples in a single channel of input data.

- SWAP: a macro used in calculating a FFT.

The following program functions are adaptations of those found in Reference 11:

- polint: performs the polynomial interpolation.

- realft: calculates Fast Fourier Transforms.

- four1: calculates Fast Fourier Transforms

- vector: allocates memory for one dimensional arrays.
- matrix: allocates memory for two dimensional arrays.
- free_matrix: deallocates memory from two dimensional arrays.
- ExitOnError: provides abnormal program termination.

2. Beamformer Source Code

```
/****
*  PROGRAM: BEAMFORMER vsn 1.0
*  WRITTEN BY: Steven Crocker
*  LAST UPDATE: October 9, 1991
*
*  This program takes input from various data files and the user.  It
*  outputs a data file.  The inputs are a number of channels of digital
*  acoustic  data, and information regarding the physical characteristics
*  and geometry of the receiving array.  Additionally, environmental data
*  in the form of normal mode eigenfunctions and eigenvalues at the
*  receiving array are required to operate this beamformer.  The output
*  is a single channel of acoustic data.
*
****/
#define UNIX VERSION            /* either ANSI or UNIX                */
#define ASCII ON                /* select output mode ON or OFF       */
#define BINARY OFF              /* select output mode ON or OFF       */
#define SIGNAL ON               /* either ON or OFF                   */
#define SPECTRUM ON             /* either ON or OFF                   */
#define LOWER_SENSOR OFF        /* either ON or OFF                   */
#define VALIDATE OFF            /* either ON or OFF                   */
#define ERROR_ESTIMATE OFF      /* either ON or OFF                   */
#define ON 1                    /* logical "switch"                   */
#define OFF 0                   /* logical "switch"                   */
#define INTERPOLATE polint      /* polint                             */
#define ORDER 3                 /* order of interpolator         (odd)*/
#define STEP 1                  /* number of steps for derivatives    */
#define TINY 1.0e-25            /* prevents division by zero          */
#define PI 3.14159265359        /* for freq to omega conversions      */
#define RADIAN 57.2957795131    /* for degree to radian conversions   */
#define OFFSET 4.0              /* dist btwn upper sensor and phone #1 */
#define DELTA_R 45.0            /* array element spacing              */
#define CTD_OFFSET 0.5          /* diff btwn ctd depth inc & 1st depth */
#define SSP_LENGTH 2500         /* max number of pts in eigunfunction */
#define EIGVAL_LENGTH 230       /* max number of eigenvalues          */
#define LOOK_DIRECTION  217.0   /* direction from which signal arrives */
#define TILT_BUFFER 120         /* max length of tilt data vectors    */
#define BUFFER_TIME 12          /* input buffer length in seconds (int)*/
#define F_SAMPLE 228            /* sampling frequency            (int)*/
#define CHANNELS 32             /* number of channels processed       */
#define F_CARRIER 57.0          /* carrier frequency                  */
#define FFT_LENGTH 2048         /* radix 2 <= (BUFFER_TIME-2)*F_SAMPLE */
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

#include<stdio.h>
#include<malloc.h>
#include<math.h>

#if defined ( ANSI )
#include<float.h>
```

```
#include<stdlib.h>
int getInput(void);
int putOutput(void);
int processTilt(float **x, float **y, float **z);
int processModes(float **z, float *weight, float *ptrC);
int dydx(float *x, float *y, float *ddx, int points);
int polint(float *xa, float *ya, int n, float x, float *y, float *dy);
int realft(float *data, int n, int isign);
int four1(float *data, int nn, int isign);
float *vector(int length);
float **matrix(int row, int col);
int free_matrix(float **m, int row);
void ExitOnError(char error_txt[]);
#elif defined ( UNIX )
int   vector(),
      matrix();
      getInput(),
      putOutput(),
      processTilt(),
      processModes(),
      dydx(),
      polint(),
         realft(),
         four1(),
      free_matrix(),
      ExitOnError();
#endif

   /* Global Variable Declarations */
float **inSOUND, *outSOUND, *MeanSqError, Max=0.0, Min=1.0e25;
int lastMinute, Minute=1, firstBuffer=1;

main()
{
   FILE *fpV1;

       int i, j, k, n, *shift;

   float **x, **y, **z, *indx, *samples, arg, ans, err, *delay,
             *delta, *weight, *pwrSpectrum, Cgroup;


   /* Memory Allocation and Tilt Data Processing */
   x=(float**)matrix(CHANNELS, TILT_BUFFER);
   y=(float**)matrix(CHANNELS, TILT_BUFFER);
   z=(float**)matrix(CHANNELS, TILT_BUFFER);
   processTilt(x, y, z);

   if (VALIDATE==ON)
   {
      printf("\nDumping array geometry to array.dat\n");

      fpV1=fopen("array.dat","wt");
      if(fpV1==NULL) ExitOnError("Error opening validation file");

      fprintf(fpV1,"Channel\t\t X\t\t Y\t\t Z\n");

      for (i=1;i<=lastMinute;i++)
      {
         fprintf(fpV1, "MINUTE: %i\n", i);
         for (j=1;j<=CHANNELS;j++)
            fprintf(fpV1,"%i\t %f\t %f\t %f\n",j,x[j][i],y[j][i],z[j][i]);
      } /* for */
      fclose(fpV1);
    } /* if */

   /* Memory Allocation*/
   weight=(float*)vector(CHANNELS);
```

```
indx=(float*)vector(ORDER+1);
delay=(float*)vector(CHANNELS);
delta=(float*)vector(CHANNELS);
outSOUND=(float*)vector((BUFFER_TIME-1)*F_SAMPLE);
inSOUND=(float**)matrix(CHANNELS, BUFFER_TIME*F_SAMPLE);
if ((shift=(int*)malloc((CHANNELS+1)*sizeof(int)))==NULL)
   ExitOnError("Memory allocation failure for shift[].");
for (i=1; i<=ORDER+1; i++) indx[i]=(float)i;
if (ERROR_ESTIMATE==ON)
   MeanSqError=(float*)vector((BUFFER_TIME-1)*F_SAMPLE);
    if (SPECTRUM==ON) pwrSpectrum=(float*)vector(FFT_LENGTH/2);
while(Minute<=lastMinute)
{

   /* Mode Data Processing */
   processModes(z, weight, &Cgroup);

   /* Calculate delays, shifts, etc */
   for (i=1; i<=CHANNELS; i++)
   {
      delay[i]=x[i][Minute]/Cgroup;             /* Time delay   */
      shift[i]=(int)(delay[i]*(float)F_SAMPLE); /* # of samples */
               /* fraction of 1 sample */
      delta[i]=delay[i]*(float)F_SAMPLE-(float)shift[i];
   } /* for */

   if (VALIDATE==ON)
   {
      printf("Dumping element delays to delay.dat\n");

      if (firstBuffer)
      {
         if ((fpV1=fopen("delay.dat", "wt")) == NULL)
            ExitOnError("Error opening validation file.");
       } /* if */
      else
      {
         if ((fpV1=fopen("delay.dat", "at")) == NULL)
            ExitOnError("Error opening validation file.");
      } /* else */

      fprintf(fpV1,"MINUTE: %i\n", Minute);

      fprintf(fpV1,
             "Channel\t delay\t\t int shift\t fraction of 1 shift\n");

      for (i=1;i<=CHANNELS;i++)
      {
         fprintf(fpV1, "%i\t %e\t %i\t% e\n",
                i,delay[i],shift[i],delta[i]);
      } /* for */

      fclose(fpV1);

      printf("Dumping phone weights and grp speed to modal.dat\n");

      if(firstBuffer)
      {
         if ((fpV1=fopen("modal.dat", "wt")) == NULL)
            ExitOnError("Error opening validation file.\n");
         fprintf(fpV1,"Group speed for F_CARRIER is: %g\n\n\n",Cgroup);
      } /* if */
      else
      {
         if ((fpV1=fopen("modal.dat","at")) == NULL)
            ExitOnError("Error opening validation file.\n");
         fprintf(fpV1,"\nHydrophone weights for minute %i\n",Minute);
      } /* else if */
```

```
   fprintf(fpV1, "Channel \tWeight\n");

   for (i=1;i<=CHANNELS;i++)
      fprintf(fpV1, "%i\t%e\n",i,weight[i]);

   fclose(fpV1);
} /* if */

       if (SPECTRUM==ON)
            for (k=1;k<=FFT_LENGTH;k++) pwrSpectrum[k]=0.0;

for (n=1; n<=60/(BUFFER_TIME-2); n++)
{
   getInput();

   if(firstBuffer)   /* Produce first output buffer */
   {
      for (i=1; i<=F_SAMPLE; i++)
      {
         outSOUND[i]=0.0;
         if (ERROR_ESTIMATE==ON) MeanSqError[i]=0.0;
      } /* for */
      for (i=F_SAMPLE+1; i<=(BUFFER_TIME-1)*F_SAMPLE; i++)
      {
         outSOUND[i]=0.0;
         if (ERROR_ESTIMATE==ON) MeanSqError[i]=0.0;

         for (j=1; j<=CHANNELS; j++)
         {
            if (delay[j]>=0.0)
            {
               arg=(float)(ORDER+1)/2.0+(1-delta[j]);
               samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2];
            } /* if */
            else if (delay[j]<0.0)
            {
               arg=(float)(ORDER+1)/2.0+delta[j];
               samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2+1];
            } /* else if */
            INTERPOLATE(indx, samples, ORDER+1, arg, &ans, &err);
            outSOUND[i]=outSOUND[i]+ans;
            if (ERROR_ESTIMATE==ON)
               MeanSqError[i]=MeanSqError[i]+err*err;
         } /* for */
         if(fabs(outSOUND[i])>Max) Max=fabs(outSOUND[i]);
         if(fabs(outSOUND[i])<Min) Min=fabs(outSOUND[i]);
         if (ERROR_ESTIMATE==ON)
            MeanSqError[i]=MeanSqError[i]/(float)CHANNELS;
      } /* for */
   } /* if */
   else      /* Produce subsequent output buffers */
   {
      for (i=F_SAMPLE+1; i<=(BUFFER_TIME-1)*F_SAMPLE; i++)
      {
         outSOUND[i-F_SAMPLE]=0.0;
         if (ERROR_ESTIMATE==ON) MeanSqError[i-F_SAMPLE]=0.0;

         for (j=1; j<=CHANNELS; j++)
         {
            if (delay[j]>=0.0)
            {
               arg=(float)(ORDER+1)/2.0+(1-delta[j]);
               samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2];
            } /* if */
            else if (delay[j]<0.0)
            {
               arg=(float)(ORDER+1)/2.0+delta[j];
               samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2+1];
```

```
                } /* else if */
                INTERPOLATE(indx, samples, ORDER+1, arg, &ans, &err);
                outSOUND[i-F_SAMPLE]=outSOUND[i-F_SAMPLE]+ans;
                if (ERROR_ESTIMATE==ON)
                 MeanSqError[i-F_SAMPLE]=MeanSqError[i-F_SAMPLE]+err*err;
              } /* for */
              if(fabs(outSOUND[i-F_SAMPLE])>Max)
                 Max=fabs(outSOUND[i-F_SAMPLE]);

              if(fabs(outSOUND[i-F_SAMPLE])<Min)
                 Min=fabs(outSOUND[i-F_SAMPLE]);
              if (ERROR_ESTIMATE==ON)
                 MeanSqError[i-F_SAMPLE]=MeanSqError[i-F_SAMPLE]/
                    (float)CHANNELS;
           } /* for */
         } /* else */
                 if (SIGNAL==ON) putOutput();

                 if (SPECTRUM==ON)
                 {
                        window(outSOUND, FFT_LENGTH);
                        realft(outSOUND, FFT_LENGTH/2, 1);
                        for (k=0;k<FFT_LENGTH;k += 2)
                        {
                 pwrSpectrum[k/2+1]=pwrSpectrum[k/2+1]+
                 outSOUND[k]*outSOUND[k]+outSOUND[k+1]*outSOUND[k+1];
                        } /* for */
                 } /* if */
                 firstBuffer=0;  /* set firstBuffer to false */
      } /* for */

             if (SPECTRUM==ON)
             {
                  for (k=1; k<=FFT_LENGTH; k++)
                     pwrSpectrum[k]=pwrSpectrum[k]*(float)(BUFFER_TIME-2)/60.0;

                  dumpSpectrum(pwrSpectrum);
             } /* if */
             printf("\t%i minutes of input data processed.\n", Minute);
        Minute++;       /* increment minute counter */
    } /* while */
    printf("EXECUTION COMPLETE: End of tilt data encountered\n");
    printf("Maximum magnitude encountered was: %e\n",Max);
    printf("Minumum magnitude encountered was: %e\n",Min);
    exit(0);
}/*********************************************************************/
 /************************** END main *******************************/
 /*********************************************************************/
/*****
* FUNCTION: getInput()
*
*  This function handles all acoustic input.  It also provides one of
*  two normal process terminations available in the program.  (The other
*  is located in main().)
*
*  Arguments:                     none
*
*  Return value:                  0
*
*  Functions called:              vector()          ExitOnError()
*
*  Definitions called:            ANSI              UNIX
*                                 F_SAMPLE          CHANNELS
*                                 BUFFER_TIME
*
*  Global variables called:       inSOUND[][]        Min
*                                 firstBuffer        Max
*
```

```
*  Significant memory allocation:   diskBuffer[]
*
*****/
#if defined ( ANSI )
int getInput(void)
#elif defined ( UNIX )
getInput()
#endif
{
   int i, j, buffer, items;
   float *diskBuffer;
   char fileName[80];
   static FILE *fpStatic;

   if (firstBuffer)
   {
      printf("Enter file name for input acoustic data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fpStatic=fopen(fileName, "rb")) == NULL)
         ExitOnError("Error opening INPUT ACOUSTIC data file.");
   } /* if */

   /* Memory Allocation */
   if(firstBuffer) buffer=BUFFER_TIME*F_SAMPLE*CHANNELS;
   else buffer=(BUFFER_TIME-2)*F_SAMPLE*CHANNELS;
   diskBuffer=(float*)vector(buffer);

   items=fread((char*)(diskBuffer+1), sizeof(float), buffer, fpStatic);
   if (items==buffer) /*continue*/;
   else if(ferror(fpStatic) != 0)
      ExitOnError("Error encountered while reading input acoustic data");
   else if(feof(fpStatic) != 0)
   {
      printf("\n\t**************************************************\n");
      printf("\t    End of File reached: EXECUTION COMPLETE\n");
      printf("\t\t%i minutes of data processed\n",Minute-1);
      printf("\t\t%i bytes of data discarded\n", items*sizeof(float));
      printf("\t\tMaximum magnitude encountered was: %e\n",Max);
      printf("\t\tMinimum magnitude encountered was: %e\n",Min);
      printf("\t    End of File reached: EXECUTION COMPLETE\n");
      printf("\t**************************************************\n");
      fclose(fpStatic);
      exit(0);
   } /* else if */
   else ExitOnError("Unknown error handling acoustic input file.");

   for (i=1; i<=BUFFER_TIME*F_SAMPLE; i++)
   {
      for (j=1; j<=CHANNELS; j++)
      {
         if (firstBuffer)
         {
            inSOUND[j][i] = diskBuffer[CHANNELS*(i-1)+j];
         } /* if */
         else
         {
            if(i<=2*F_SAMPLE)
               inSOUND[j][i]=inSOUND[j][i+(BUFFER_TIME-2)*F_SAMPLE];
            else
               inSOUND[j][i]=diskBuffer[CHANNELS*(i-2*F_SAMPLE-1)+j];
         } /* else */
      } /* for */
   } /* for */
```

```
   /* Deallocate Memory */
   free((char*)diskBuffer);
   return( 0 );
} /************************/
  /***** END getInput *****/
  /************************/
/*****
* FUNCTION: putOutput()
*
*  This function handles all acoustic output.  Additionally, it outputs
*  the estimated mean squared error from the interpolators (if enabled).
*
*  Arguments:                    none
*
*  Return value:                 0
*
*  Functions called:             ExitOnError()
*
*  Definitions called:           ANSI              UNIX
*                                F_SAMPLE          BUFFER_TIME
*
*  Global variables called:      outSOUND[]        MeanSqError[]
*                                firstBuffer
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
int putOutput(void)
#elif defined ( UNIX )
putOutput()
#endif
{
   int i, cut;
   char fileName[12], mode[2];
   static FILE *fpOutSound, *fpMSE;

   if(firstBuffer)
   {
      if (ASCII==ON)
      {
          mode[0]='w';
          mode[1]='t';
      } /* if */
      else if (BINARY==ON)
      {
         mode[0]='w';
         mode[1]='b';
      } /* else if */

      cut=1;
      printf("Enter file name for output data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fpOutSound=fopen(fileName, mode)) == NULL)
         ExitOnError("Error opening OUTPUT data file.");

      if (ERROR_ESTIMATE==ON)
      {
         printf("Opening file error.dat\n\n");

         if ((fpMSE=fopen("error.dat", mode)) == NULL)
            ExitOnError("Error opening error.dat");
      } /* if */
```

```
   } /* if */
   else cut=2;

   if (ERROR_ESTIMATE==ON)
   {
      if (ASCII==ON)
      {
         for (i=1; i<=(BUFFER_TIME-cut)*F_SAMPLE; i++)
            fprintf(fpMSE,"%e\n", MeanSqError[i]);
      } /* if */
      else if (BINARY==ON)
      {
         if(fwrite((char*)(MeanSqError+1),sizeof(float),(BUFFER_TIME-cut)*
            F_SAMPLE,fpMSE)==(unsigned)(BUFFER_TIME-cut)*F_SAMPLE) ;
         else if(ferror(fpMSE) != 0)
            ExitOnError("Error encountered writing error data");
         else ExitOnError("Unknown error handling error file.");
      } /* else if */
   } /* if */

   if (ASCII==ON)
   {
      for (i=1;i<=(BUFFER_TIME-cut)*F_SAMPLE; i++)
         fprintf(fpOutSound, "%e\n", outSOUND[i]);
   } /* if */
   else if (BINARY==ON)
   {
      if(fwrite((char*)(outSOUND+1),sizeof(float),(BUFFER_TIME-cut)*
         F_SAMPLE,fpOutSound)==(unsigned)(BUFFER_TIME-cut)*F_SAMPLE) ;
      else if(ferror(fpOutSound) != 0)
         ExitOnError("Error encountered writing output acoustic data");
      else ExitOnError("Unknown error handling acoustic output file.");
   } /* else if */
   return( 0 );
} /*************************/
  /***** END putOutput *****/
  /*************************/
/*****
* FUNCTION: dumpSpectrum()
*
*  This function handles dumps the signal power spectrum (if selected).
*
*  Arguments:                    pwrSpectrum
*
*  Return value:                 0
*
*  Functions called:             ExitOnError()
*
*  Definitions called:           ANSI              UNIX
*                                F_SAMPLE          FFT_LENGTH
*
*  Global variables called:      none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
int dumpSpectrum ( float *pwrSpectrum )
#elif defined ( UNIX )
dumpSpectrum ( pwrSpectrum )
float *pwrSpectrum;
#endif
{
      int i;
      static float sequence=1.0;
      char fileName[12], mode[2];
      static FILE *fp;
```

```
      if(firstBuffer)
      {
            if (ASCII==ON)
            {
                  mode[0]='w';
                  mode[1]='t';
            } /* if */
            else if (BINARY==ON)
            {
                 mode[0]='w';
                 mode[1]='b';
            } /* else if */

            printf("Enter file name for output SPECTRUM data: ");

            if((scanf("%s", fileName))==EOF)
                 ExitOnError("Fatal error in scanf()");
            printf("\n\n");

            if ((fp=fopen(fileName, mode)) == NULL)
                 ExitOnError("Error opening OUTPUT SPECTRUM data file.");

            printf("Select output format: \n");
            printf("   Enter 0 for MATLAB compatible output.\n");
            printf("   Enter 1 for GRAFTOOL compatible output.\n");
            if((scanf("%i", &i))==EOF)
                 ExitOnError("Fatal error in scanf()");
            if((i != 0) && (i != 1)) ExitOnError("Invalid output selection");
      } /* if */

      if (ASCII==ON)
      {
            if(firstBuffer && format==1)
            {
                 fprintf(fp,"0 ");
                 for (i=1, i<=FFT_LENGTH/2; i++)
                       fprintf(fp,"%e ",(float)(i-1)*F_SAMPLE/FFT_LENGTH);
                 fprintf(fp,"\n");
            } /* if */

            if (format==1) fprintf(fp,"%e ",sequence);

            for (i=1;i<=FFT_LENGTH/2; i++)
                 fprintf(fp,"%e ", pwrSpectrum[i]);

            fprintf(fp,"\n ");
            sequence=sequence+1.0;
      } /* if */
      else if (BINARY==ON)
      {
            if(fwrite((char*)(pwrSpectrum+1),sizeof(float),FFT_LENGTH/2,fp)==
                 (unsigned)FFT_LENGTH/2) ;
            else if(ferror(fpOutSound) != 0)
               ExitOnError("Error encountered writing output SPECTRUM data");
     else ExitOnError("Unknown error handling acoustic SPECTRUM file.");
      } /* else if */
      return( 0 );
} /*****************************/
  /***** end dumpSpectrum *****/
  /*****************************/
/*****
* FUNCTION: processTilt()
*
*  This function handles all array tilt data.  It calculates the X, Y, Z
*  coordinates of each hydrophone as a function of time.  The coordinate
*  system is oriented such that X points toward the signal "origin" and
*  Z points down.
*
```

```
*  Arguments:                      x[][]          y[][]
*                                  z[][]
*
*  Return value:                   0
*
*  Functions called:               ExitOnError()  matrix()
*                                  vector()       free_matrix()
*
*  Definitions called:             ANSI           UNIX
*                                  DELTA_R        LOWER_SENSOR
*                                  RADIAN         CHANNELS
*                                  LOOK_DIRECTION OFFSET
*                                  TILT_BUFFER
*
*  Global variables called:        lastMinute
*
*  Significant memory allocation:  xx[][]         yy[][]
*                                  zz[][]         tilt[]
*                                  angle[]        udepth[]
*                                  ldepth[]
*
*****/
#if defined ( ANSI )
int processTilt(float **x, float **y, float **z)
#elif defined ( UNIX )
processTilt(x,y,z)
float **x, **y, **z;
#endif
{
   int i, j, notEOF;
   float *tilt, *angle, *udepth, *ldepth, **xx, **yy, **zz, theta;
   char fileName[12];
   FILE *fp1, *fp2;

   /* Open Data Files */
   printf("Enter file name for upper tilt sensor data: ");

   if((scanf("%s", fileName))==EOF)
      ExitOnError("Fatal error in scanf()");

   printf("\n\n");

   if ((fp1=fopen(fileName, "rt")) == NULL)
      ExitOnError("Error opening UPPER TILT data file.");

   if (LOWER_SENSOR==ON)
   {
      printf("Enter file name for lower tilt sensor data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp2=fopen(fileName, "rt")) == NULL)
         ExitOnError("Error opening LOWER TILT data file.");
      ldepth=(float*)vector(TILT_BUFFER);
   } /* if */

   /* Memory Allocation */
   tilt=(float*)vector(TILT_BUFFER);
   angle=(float*)vector(TILT_BUFFER);
   udepth=(float*)vector(TILT_BUFFER);
   xx=(float**)matrix(CHANNELS, TILT_BUFFER);
   yy=(float**)matrix(CHANNELS, TILT_BUFFER);
   zz=(float**)matrix(CHANNELS, TILT_BUFFER);

   i=1;                    /* read upper tilt data */
```

```
   notEOF=1;
   while(notEOF)
   {
    if(fscanf(fp1,"%g %g %g\n",&tilt[i],&angle[i],&udepth[i]) != EOF) i++;
      else notEOF=0;
   }

   if (LOWER_SENSOR==ON)
   {
      j=1;                  /* read lower tilt data */
      notEOF=1;
      while(notEOF)
      {
         if(fscanf(fp2, "%g\n",&ldepth[j]) != EOF) j++;
         else notEOF=0;
      } /* while */
      if (i<=j) lastMinute=i-1;
      else lastMinute=j-1;
   }
   else lastMinute=i-1;

   /************This is the assumed array geometry: LINEAR**********/
   for (j=1; j<=CHANNELS; j++)
   {
      for (i=1; i<=lastMinute; i++)
      {
         xx[j][i]=DELTA_R*(float)(j-1)*cos(angle[i]/RADIAN)*
            sin(tilt[i]/RADIAN);
         yy[j][i]=DELTA_R*(float)(j-1)*sin(angle[i]/RADIAN)*
            sin(tilt[i]/RADIAN);
         zz[j][i]=DELTA_R*(float)(j-1)*cos(tilt[i]/RADIAN)+
            OFFSET*cos(tilt[i]/RADIAN)+udepth[i];
      } /* for */
   } /* for */
   /*********************************************************************/

   theta=(360.0-LOOK_DIRECTION)/RADIAN; /* coordinate rotation  */
   for (j=1; j<=CHANNELS; j++)
   {
      for (i=1; i<=lastMinute; i++)    /* points x into signal */
      {
         x[j][i]=xx[j][i]*cos(theta)-yy[j][i]*sin(theta);
         y[j][i]=xx[j][i]*sin(theta)+yy[j][i]*cos(theta);
         z[j][i]=zz[j][i];
      } /* for */
   } /* for */

   /* Memory Deallocation */
   if (LOWER_SENSOR==ON) free((char*)ldepth);
   free((char*)tilt);

   free((char*)angle);
   free((char*)udepth);
   free_matrix(xx, CHANNELS);
   free_matrix(yy, CHANNELS);
   free_matrix(zz, CHANNELS);
   fclose(fp1);
   if (LOWER_SENSOR==ON) fclose(fp2);
   return( 0 );
} /***************************/
  /***** END processTilt *****/
  /***************************/
/*****
* FUNCTION: processModes()
*
*  This function handles the normal mode data.  It calculates hydrophone
*  weights and group speed.  The user must insure that the depth vector
*  and eigenfunction vector are of equal length.
```

```
*
*  Arguments:                       z[][]           weight[]
*                                   ptrC
*
*  Return value:                    0
*
*  Functions called:                vector()        ExitOnError()
*                                   INTERPOLATE()   dydx()
*
*  Definitions called:              ANSI            UNIX
*                                   PI              SSP_LENGTH
*                                   EIGVAL_LENGTH   ORDER
*                                   CHANNELS        F_CARRIER
*
*  Global variables called:         Minute
*
*  Significant memory allocation:   depth[]         Zn[]
*                                   OnOff[]         w[]
*                                   Kr[]            dwdK
*
*****/
#if defined ( ANSI )
int processModes(float **z, float *weight, float *ptrC)
#elif defined ( UNIX )
processModes(z,weight,ptrC)
float **z, *weight, *ptrC;
#endif
{
   int i, j, ptsEigVal, set, notEOF, deadPhones, weightNotAssigned;
   static int ptsEigFun;
   float  *w, *Kr, *dwdK, err;
   static float depth[SSP_LENGTH+1], Zn[SSP_LENGTH+1], OnOff[CHANNELS+1];
   char key, fileName[12];
   FILE *fp1, *fp2;

   if(firstBuffer==1)
   {
      w=(float*)vector(EIGVAL_LENGTH);
      Kr=(float*)vector(EIGVAL_LENGTH);
      dwdK=(float*)vector(EIGVAL_LENGTH);

      printf("Enter file name for normal mode data (eigenfunction): ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp1=fopen(fileName, "rt")) == NULL)
       ExitOnError("Error opening NORMAL MODE data file (eigenfunction)");

      printf("Enter file name for normal mode data (eigenvalues): ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp2=fopen(fileName, "rt")) == NULL)
        ExitOnError("Error opening NORMAL MODE data file (eigenvalues).");

      i=1;                    /* read normal mode data */
      notEOF=1;
      while(notEOF)
      {
         if(fscanf(fp1, "%g %g \n", &depth[i], &Zn[i]) != EOF) i++;
         else notEOF=0;
      }
```

```
   ptsEigFun=i-1;

   for(i=1;i<=ptsEigFun;i++) depth[i]=depth[i]+CTD_OFFSET;

   i=1;
   notEOF=1;
   while(notEOF)
   {
      if(fscanf(fp2, "%g %g \n", &w[i], &Kr[i]) != EOF) i++;
      else notEOF=0;
   }
   ptsEigVal=i-1;

   for (i=1;i<=CHANNELS;i++) OnOff[i]=1.0;

   printf("Do you want to turn off any hydrophones? ");

   if((scanf("%s",&key))==EOF)
      ExitOnError("Fatal error in scanf()");

   if(key=='y' || key=='Y')
   {
      printf("\nHow many hydrophones must be secured? ");

      if((scanf("%i", &deadPhones))==EOF)
         ExitOnError("Fatal error in scanf()");

      for(i=1;i<=deadPhones;i++)
      {
         printf("\nEnter hydrophone number to secure:  ");

         if((scanf("%i", &j))==EOF)
            ExitOnError("Fatal error in scanf()");

         if (j>CHANNELS || j<1)
            ExitOnError("Bad hydrophone identification");
         OnOff[j]=0.0;
      } /* for */
   } /* if */
} /* if */

for(i=1;i<=lastMinute;i++)
{
   if(depth[ptsEigFun]<z[CHANNELS][i])
   {
      printf("Max eigenfunction depth is: %f\n",depth[ptsEigFun]);
      printf("at depth[] index of: %i\n",ptsEigFun);
      printf("Max depth of phone number %i is: %f\n\n",
          CHANNELS,z[CHANNELS][i]);
      ExitOnError("Fatal data set error");
   } /* if */
} /* for */

j=1;
for (i=1; i<=CHANNELS; i++)
{
   weightNotAssigned=1;
   while (j<=ptsEigFun && weightNotAssigned)
   {
      if(z[i][Minute]<0.0 || depth[j]<0.0)
      {
         printf("i=%i\n",i);
         printf("j=%i\n",j);
         printf("Minute=%i\n",Minute);
         printf("z[i][Minute] is: %f\n", z[i][Minute]);
         printf("depth[j] is: %f\n",depth[j]);
         printf("Depth less than zero encountered in processModes.");
         printf("\n\n");
```

```
            ExitOnError("Check input depths for coordinate orientation");
         } /* if */


         if(depth[j]<z[i][Minute] && depth[j+1]>z[i][Minute])
         {
            set=(ORDER+1)/2;
            INTERPOLATE(&depth[j-set], &Zn[j-set], ORDER+1, z[i][Minute],
               &weight[i], &err);
            weightNotAssigned=0;
         } /* if */
         j++;
      } /* while */
   } /* for */

   for (i=1; i<=CHANNELS; i++) weight[i]=OnOff[i]*weight[i];

   if (Minute==1)
   {
      dydx(Kr,w,dwdK,ptsEigVal);
      for (i=1;i<=ptsEigVal;i++)
      {
         if (w[i]<2.0*PI*F_CARRIER && w[i+1]>2.0*PI*F_CARRIER)
         {
            set=(ORDER+1)/2;
            INTERPOLATE(&w[i-set],&dwdK[i-set],ORDER+1,
                    2.0*PI*F_CARRIER,ptrC,&err);
         } /* if */
      } /* for */

      free((char*)w);
      free((char*)Kr);
      free((char*)dwdK);
   } /* if */
   return( 0 );
} /****************************/
  /***** END processModes *****/
  /****************************/
/*****
* FUNCTION: dydx()
*
*  This function estimates derivatives.
*
*  Arguments:                       x[]              y[]
*                                   ddx[]            points
*
*  Return value:                    0
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI             UNIX
*                                   STEP
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
int dydx(float *x, float *y, float *ddx, int points)
#elif defined ( UNIX )
dydx(x,y,ddx,points)
float *x, *y, *ddx;
int points;
#endif
{
   int n;
```

```
   for (n=1;n<=points;n++)
   {
      if ((n>=STEP) && (n<=points-STEP))  /*center*/
         ddx[n]=(y[n+STEP]-y[n-STEP])/(x[n+STEP]-x[n-STEP]);

      else if (n<STEP)  /*beginning*/
         ddx[n]=(y[n+STEP]-y[1])/(x[n+STEP]-x[1]);

      else if (n>points-STEP)  /*end*/
         ddx[n]=(y[points]-y[n-STEP])/(x[points]-x[n-STEP]);

      else
         ExitOnError("Index error in dydx");  /* sanity check */
   } /* for */
   return( 0 );
} /********************/
  /***** END dydx *****/
  /********************/
/*****
* FUNCTION: polint()
*
*  This function performs polynomial interpolation.
*
*  Arguments:                      xa[]            ya[]
*                                  n               x
*                                  y               dy
*
*  Return value:                   0
*
*  Functions called:               vector()        ExitOnError()
*
*  Definitions called:             ANSI            UNIX
*
*  Global variables called:        none
*
*  Significant memory allocation:  d[]             c[]
*
*****/
#if defined ( ANSI )
int polint( float *xa, float *ya, int n, float x, float *y, float *dy)
#elif defined ( UNIX )
polint(xa,ya,n,x,y,dy)
float *xa, *ya, x, *y, *dy;
int n;
#endif
{
   int i, m, ns=1;
   float den, dif, dift, ho, hp, w;
   float *c, *d;

   dif=fabs(x-xa[1]);

   c=(float*)vector(n);
   d=(float*)vector(n);

   for (i=1; i<=n; i++)
   {
      if ((dift=fabs(x-xa[i])) < dif)
      {
         ns=i;
         dif=dift;
      } /* if */
      c[i]=ya[i];
      d[i]=ya[i];
   } /* for */
   *y=ya[ns--];
   for (m=1; m<n; m++)
   {
```

```
      for (i=1; i<=n-m; i++)
      {
         ho=xa[i]-x;
         hp=xa[i+m]-x;
         w=c[i+1]-d[i];
        if ((den=ho-hp)==0.0)
          ExitOnError("Error in routine POLINT");
         den=w/den;
         d[i]=hp*den;
         c[i]=ho*den;
      } /* for */
      *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
   } /* for */
   free((char*)d);
   free((char*)c);
   return( 0 );
} /*********************/
  /***** END polint ****/
  /*********************/
/*****
* FUNCTION: window()
*
*  This function applies a Blackman window to a vector.
*
*  Arguments:                       data[]          N
*
*  Return value:                    0
*
*  Functions called:                none
*
*  Definitions called:              ANSI            UNIX
*                                   PI
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/

#if defined ( ANSI )
int window(float *data, int N);
#elif defined ( UNIX )
window( data, N )
float *data;
int N;
#endif
{
       int n;

       for (n=0; n<N; n++)
       {
              data[n+1]=data[n+1]*(0.42+0.5*cos(2.0*PI*(float)(n-N/2)/(float)(N-1))
                   +0.08*cos(4.0*PI*(float)(n-N/2)/(float)(N-1)));
       } /* for */
       return ( 0 );
} /*********************/
  /***** end window *****/
  /*********************/
/*****
* FUNCTION: realft()
*
*  This function calculates FFT's
*
*  Arguments:                       data[]          n
*                                   isign
*
*  Return value:                    0
*
```

```
*  Functions called:                four1()
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
int realft(float *data, int n, int isign)
#elif defined ( UNIX )
realft(data, n, isign)
float *data;
int n, isign;
#endif
{
       int i, i1, i2, i3, i4, n2p3;
       float c1=0.5, c2, h1r, h1i, h2r, h2i;
       double wr, wi, wpr, wpi, wtemp, theta;

       theta=3.141592653589793/(double)n;
       if (isign==1)
       {
              c2 = -0.5;
              four1(data, n, 1);
       } /* if */
       else
       {
              c2=0.5;
              theta = -theta;
       } /* else */
       wtemp=sin(0.5*theta);
       wpr = -2.0*wtemp*wtemp;
       wpi=sin(theta);
       wr=1.0+wpr;
       wi=wpi;
       n2p3=2*n+3;
       for (i=2; i<=n/2; i++)
       {
              i4=1+(i3=n2p3-(i2=1+(i1=i+i-1)));
              h1r=c1*(data[i1]+data[i3]);
              h1i=c1*(data[i2]-data[i4]);
              h2r = -c2*(data[i2]+data[i4]);
              h2i=c2*(data[i1]-data[i3]);
              data[i1]=h1r+wr*h2r-wi*h2i;
              data[i2]=h1i+wr*h2i+wi*h2r;
              data[i3]=h1r-wr*h2r+wi*h2i;
              data[i4] = -h1i+wr*h2i+wi*h2r;
              wr=(wtemp=wr)*wpr-wi*wpi+wr;
              wi=wi*wpr+wtemp*wpi+wi;
       } /* for */
       if (isign==1)
       {
              data[1]=(h1r=data[1])+data[2];
              data[2]=h1r-data[2];
       } /* if */
       else
       {
              data[1]=c1*((h1r=data[1])+data[2]);
              data[2]=c1*(h1r-data[2]);
              four1(data,n,-1);
       } /* else */
       return ( 0 );
} /**********************/
  /***** end realft *****/
  /**********************/
/*****
```

```
* FUNCTION: four1()
*
*  This function calculates FFT's
*
*  Arguments:                      data[]          nn
*                                  isign
*
*  Return value:                   0
*
*  Functions called:               none
*
*  Definitions called:             ANSI            UNIX
*                                  SWAP
*
*  Global variables called:        none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
int four1(float *data, int nn, int isign)
#elif defined ( UNIX )
four1(data, nn, isign)
float *data;
int nn, isign;
#endif
{
      int n, mmax, m, j, istep, i;
      double wtemp, wr, wpr, wpi, wi, theta;
      float tempr, tempi;

      n=nn << 1;
      j=1;
      for (i=1; i<n; i+=2)
      {
            if (j > 1)
            {
                  SWAP(data[j],data[i]);
                  SWAP(data[j+1],data[i+1]);
            } /* for */
            m=n >> 1;
            while (m >= 2 && j > m)
            {
                  j -= m;
                  m >>=1;
            } /* while */
            j += m;
      } /* for */
      mmax=2;
      while (n > mmax)
      {
            istep=2*mmax;
            theta=6.28318530717959/(isign*mmax);
            wtemp=sin(0.5*theta);
            wpr = -2.0*wtemp*wtemp;
            wpi=sin(theta);
            wr=1.0;
            wi=0.0;
            for (m=1; m<mmax; m+=2)
            {
                  for (i=m; i<=n; i+=istep)
                  {
                        j=i+mmax;
                        tempr=wr*data[j]-wi*data[j+1];
                        tempi=wr*data[j+1]+wi*data[j];
                        data[j]=data[i]-tempr;
                        data[j+1]=data[i+1]-tempi;
                        data[i] += tempr;
```

```
                    data[i+1] += tempi;
                } /* for */
                wr=(wtemp=wr)*wpr-wi*wpi+wr;
                wi=wi*wpr+wtemp*wpi+wi;
            } /* for */
            mmax=istep;
        } /* while */
        return ( 0 );
} /*********************/
  /***** end four1 *****/
  /*********************/
/*****
* FUNCTION: vector()
*
*  This function allocates memory for UNIT OFFSET vectors.
*
*  Arguments:                       length
*
*  Return value:                    *v
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   v[]
*
*****/
#if defined ( ANSI )
float *vector(int length)
#elif defined ( UNIX )
vector(length)
int length;
#endif
{
   float *v;

   if ((v=(float*)malloc((length+1)*sizeof(float)))==NULL)
      ExitOnError("Memory allocation failure in vector().");
#if defined ( ANSI )
   return v;
#elif defined ( UNIX )
   return (long int)v;
#endif
} /************************/
  /***** END vector *****/
  /************************/
/*****
* FUNCTION: matrix()
*
*  This function allocates memory for UNIT OFFSET 2-D arrays.
*
*  Arguments:                       row               col
*
*  Return value:                    **m
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   m[][]
*
*****/
#if defined ( ANSI )
float **matrix(int row, int col)
```

```
#elif defined ( UNIX )
matrix(row,col)
int row, col;
#endif
{
   int i;
   float **m;

   if ((m=(float**)malloc((unsigned)(row+1)*sizeof(float*)))==NULL)
      ExitOnError("Allocation failure 1 in matrix()");
   for (i=1; i<=row; i++)
   {
      if ((m[i]=(float*)malloc((unsigned)(col+1)*sizeof(float)))==NULL)
         ExitOnError("Allocation failure 2 in matrix()");
   } /* for */
#if defined ( ANSI )
   return m;
#elif defined ( UNIX )
   return (long int)m;
#endif
} /**********************/
  /***** END matrix *****/
  /**********************/
/*****
* FUNCTION: free_matrix()
*
*  This function deallocates memory from UNIT OFFSET 2-D arrays.
*
*  Arguments:                       m[][]            row
*
*  Return value:                    0
*
*  Functions called:                none
*
*  Definitions called:              ANSI          UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
void free_matrix(float **m, int row)
#elif defined ( UNIX )
free_matrix(m,row)
float **m;
int row;
#endif
{
   int i;

   for(i=row; i>=1; i--)
      free((char*)m[i]);
   free((char*)m);
   return( 0 );
} /***************************/
  /***** END free matrix *****/
  /***************************/
/*****
* FUNCTION: ExitOnError()
*
*  This function performs an abnormal process termination.
*
*  Arguments:                       error_txt[]
*
*  Return value:                    none
*
*  Functions called:                none
```

```
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
void ExitOnError(char error_txt[])
#elif defined ( UNIX )
ExitOnError(error_txt)
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   exit(0);
} /***************************/
  /***** END ExitOnError *****/
  /***************************/
```

B. DECOMMA.C

This program removes commas and colons from the output of the upper tilt sensor of the Heard Island Array.

```
/*****
*  PROGRAM DECOMMA.C vsn 1.0
*  WRITTEN BY: Steven Crocker
*  LAST UPDATE: August 3, 1991
*
*  This program takes any ASCII file and copies it to a user
*  defined file.  It removes the commas and colons.
*****/

#define VERSION ANSI              /* either ANSI or UNIX       */
#define C 25                      /* max length for file names */
#define COMMA 44                  /* ASCII codes               */
#define COLON 58
#define SPACE 32
#include<stdio.h>
#include <malloc.h>

#if (VERSION==ANSI)
#include<stdlib.h>
void ExitOnError(char error_txt[]);
#endif

main()
{
   FILE *sacm_fp, *out_fp;
   char c, sacm_file[C], out_file[C];

   printf("\n\n\nEnter file name for input SACM data.\n");
   scanf("%s",sacm_file);
   printf("\n\n");
   printf("Enter file name for output SACM data.\n");
   scanf("%s",out_file);
   if ((sacm_fp=fopen(sacm_file, "rt")) != NULL) {
      if ((out_fp=fopen(out_file, "wt")) != NULL) {
       while((c=fgetc( sacm_fp )) != EOF) {
```

```
            if((c==COMMA) || (c==COLON)) c=SPACE;
            fputc(c, out_fp);
         } /* while */
      } /* if */
            else ExitOnError("Error opening output data file");
   } /* if */
      else ExitOnError("Error opening input data file");
      fclose(sacm_fp);
      fclose(out_fp);
   exit(0);
}

#if (VERSION==ANSI)
void ExitOnError(char error_txt[])
#elif (VERSION==UNIX)
ExitOnError(error_txt)
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   exit(1);
}
```

C. SACM1.C

This program reads the data output by the upper tilt sensor and formats it for use in the beamformer. It calculates 60 second averages for all data fields, converts pressure to depth and calculates tilt direction based on current velocity.

```
/*****
*  PROGRAM SACM1.C vsn 1.0
*  WRITTEN BY: Steven Crocker
*  LAST UPDATE: August 6, 1991
*
*  This program takes SACM data from the Heard Island West Coast
*  Array (upper instrument package) and condenses it.
*
*  Pressure is converted to depth.
*
*  The conductivity is not processed.  All values output are 60
*  second averages of the input.
*
*****/

#define VERSION ANSI              /* either ANSI or UNIX        */
#define SELECT_OUTPUT OFF         /* ON or OFF                  */
#define C 25                      /* max length for file names  */
#define inBUFFER 1200             /* input buffer size          */
#define outBUFFER 50              /* output buffer size         */
#define PI 3.14159265359
#define RADIAN 57.2957795131

#include<stdio.h>
```

```
#include <malloc.h>
#include<math.h>

#if (VERSION==ANSI)
#include<stdlib.h>
void ExitOnError(char error_txt[]);
#endif

main()
{
   FILE *sacm_fp, *out_fp;

   char c, sacm_file[C], out_file[C];

   int i, j, count, loop_count, inBUFFER_full, out_flag[6];

   float t1, avgTilt, avgPress, avgVN, avgVE, avgTemp, avgAngle,
         avgVelocity, junk;

   static float inTilt[inBUFFER], inPress[inBUFFER], inVN[inBUFFER],
         inVE[inBUFFER], inTime[inBUFFER], inTemp[inBUFFER],
         outTime[outBUFFER], outTilt[outBUFFER],
         outPress[outBUFFER], outAngle[outBUFFER],
         outVelocity[outBUFFER], outTemp[outBUFFER];

   printf("\n\n\nEnter file name for SACM data.\n");
   scanf("%s",sacm_file);
   printf("\n\n");
   printf("Enter file name for output.\n");
   scanf("%s",out_file);
   printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
   printf("The following pertains to output file format.\n\n");
      printf("Units of measure:   Time index is minutes.\n");
   printf("                    Angles in degrees\n");
   printf("                    Compass direction (TRUE)\n");
   printf("                    Velocity in meters/second\n");
   printf("                    Temperature in degrees C\n");
   printf("                    Depth in meters\n\n");
   printf("NOTE:  Output column order is same as above.\n\n\n");

      for (i=0;i<6;i++) out_flag[i]=1;

      #if (SELECT_OUTPUT==ON)
      printf("Include TIME? (y or n)\n");
      scanf("%s",&c);
      if(c==89 || c==121) out_flag[0]=1;
      else out_flag[0]=0;
      printf("\n\n");

      printf("Include TILT? (y or n)\n");
      scanf("%s",&c);
      if(c==89 || c==121) out_flag[1]=1;
      else out_flag[1]=0;
      printf("\n\n");

      printf("Include CURRENT DIRECTION? (y or n)\n");
      scanf("%s",&c);
      if(c==89 || c==121) out_flag[2]=1;
      else out_flag[2]=0;
      printf("\n\n");

      printf("Include CURRENT VELOCITY? (y or n)\n");
      scanf("%s",&c);
      if(c==89 || c==121) out_flag[3]=1;
      else out_flag[3]=0;
      printf("\n\n");

      printf("Include TEMPERATURE? (y or n)\n");
```

```
    scanf("%s",&c);
    if(c==89 || c==121) out_flag[4]=1;
    else out_flag[4]=0;
    printf("\n\n");

    printf("Include DEPTH? (y or n)\n");
    scanf("%s",&c);
    if(c==89 || c==121) out_flag[5]=1;
    else out_flag[5]=0;
    #endif

inBUFFER_full=inBUFFER;
loop_count=0;
    if ((sacm_fp=fopen(sacm_file, "rt")) != NULL) ;
    else ExitOnError("Error opening input data file");
    if ((out_fp=fopen(out_file, "wt")) != NULL) ;
    else ExitOnError("Error opening output data file");

    while (1)
    {
          for (i=0;i<inBUFFER;i++)
          {
               if (fscanf(sacm_fp, "%f %f %f %f %f %f %f\n",
                     &inTime[i],&inVN[i], &inVE[i], &inTemp[i],
                     &inTilt[i], &inPress[i], &junk) != EOF) ;
               else inBUFFER_full=i-2;
          } /* for */
          j=0;
          for (i=0;i<inBUFFER_full;i++)
          {
               t1=inTime[i];
               avgTilt=0.0;
               avgPress=0.0;
               avgVN=0.0;
               avgVE=0.0;
               avgTemp=0.0;
               count=0;
               while((inTime[i]<t1+60.0) && (i<inBUFFER_full))
               {
                     avgTilt=avgTilt + inTilt[i]/10.0;
                     avgPress=avgPress + inPress[i]*1000.0;
                     avgVN=avgVN + inVN[i]/1000.0;
                     avgVE=avgVE + inVE[i]/1000.0;
                     avgTemp=avgTemp + inTemp[i]/100.0;
                     i++;
                     count++;
               } /* while */
               avgTilt=avgTilt/(float)count;
               avgPress=avgPress/(float)count;
               avgTemp=avgTemp/(float)count;
               avgVN=avgVN/(float)count;
               avgVE=avgVE/(float)count;
               avgVelocity=sqrt(avgVN*avgVN+avgVE*avgVE);
               avgAngle=atan2(avgVE, avgVN)*RADIAN;
               if (avgAngle<=0.0) avgAngle=360.0+avgAngle;
               outTime[j]=(float)(j+1)+outBUFFER*loop_count;
               outTilt[j]=avgTilt;
               outPress[j]=(avgPress-101325.0)/(9.80665*1026.0);
               outTemp[j]=avgTemp;
               outVelocity[j]=avgVelocity;
               outAngle[j]=avgAngle;
               j++;
          } /* for */

          if (inBUFFER_full != inBUFFER) j--;
          for (i=0;i<j;i++)
          {
               if (out_flag[0]) fprintf(out_fp, "%6.0f", outTime[i]);
```

```
                if (out_flag[1]) fprintf(out_fp, "   %8.5f",  outTilt[i]);
                if (out_flag[2]) fprintf(out_fp, "  %8.3f", outAngle[i]);
                if (out_flag[3]) fprintf(out_fp, "   %8.6f", outVelocity[i]);
                if (out_flag[4]) fprintf(out_fp, "   %8.5f", outTemp[i]);
                if (out_flag[5]) fprintf(out_fp, "   %8.0f", outPress[i]);
                fprintf(out_fp, "\n");
            } /* for */

            if (inBUFFER_full != inBUFFER) exit(0);
            loop_count++;
        } /* while */
        exit(1);
}

#if (VERSION==ANSI)
void ExitOnError(char error_txt[])
#elif (VERSION==UNIX)
ExitOnError(error_txt)
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   exit(1);
}
```

D. SACM2.C

This program takes the output of sacm1.c as input. It locates and copies a user defined subset of the tilt data for use in beamforming.

```
/*****
*  PROGRAM SACM2.C vsn 1.0
*  WRITTEN BY: Steven Crocker
*  LAST UPDATE: August 6, 1991
*
*  This program takes data processed by SACM1.C and cuts
*  a user defined segment from it.  The segment is retained
*  as the output data file.  The input file is not affected.
*  The output time base may either be normalized to 1 or may
*  retain the original values.
*
*  The option is given to accept a default output format.  This
*  output format coincides with the required input format to the
*  time domain modal beamformer.
*
*  The output elements are selectable if desired.
*****/

#define VERSION ANSI              /* either ANSI or UNIX       */
#define C 25                      /* max length for file names */
#include<stdio.h>
#include <malloc.h>

#if (VERSION==ANSI)
#include<stdlib.h>
void ExitOnError(char error_txt[]);
#endif
```

```
main()
{
   FILE *input_fp, *out_fp;
       char c, Default, input_file[C], out_file[C];
       int i=0, out1, outN, out_flag[7];
       float Tilt, Depth, Time, Temp, Angle, Velocity;

       printf("\n\n\nEnter file name for input data.\n");
   scanf("%s",input_file);
       if ((input_fp=fopen(input_file, "rt")) != NULL) ;
       else ExitOnError("Error opening input file");
       printf("\n\n");

   printf("Enter file name for output.\n");
   scanf("%s",out_file);
       if ((out_fp=fopen(out_file, "wt")) != NULL) ;
       else ExitOnError("Error opening output file");

       printf("\n\n\nDo you want the default output configuration?\n");
       printf("This option formats the output for use in the beamformer.\n");
       scanf("%s",Default);
       printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
       if(Default != 'n' && Default != 'N')
       {
              printf("The following pertains to output file format.\n\n");
              printf("Units of measure:   Time Index is minutes.\n");
              printf("                    Angles in degrees\n");
              printf("                    Compass direction (TRUE)\n");
              printf("                    Velocity in meters/second\n");
              printf("                    Temperature in degrees C\n");
              printf("                    Depth in meters\n\n");
              printf("NOTE:  Output column order is same as above.\n\n\n");

              printf("Include TIME? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[0]=1;
              else out_flag[0]=0;
              printf("\n\n");

              printf("Include TILT? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[1]=1;
              else out_flag[1]=0;
              printf("\n\n");

              printf("Include CURRENT DIRECTION? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[2]=1;
              else out_flag[2]=0;
              printf("\n\n");

              printf("Include CURRENT SPEED? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[3]=1;
              else out_flag[3]=0;
              printf("\n\n");

              printf("Include TEMPERATURE? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[4]=1;
              else out_flag[4]=0;
              printf("\n\n");

              printf("Include DEPTH? (y or n)\n");
              scanf("%s",c);
              if(c==89 || c==121) out_flag[5]=1;
              else out_flag[5]=0;
       } /* if */
```

```
        else
        {
               out_flag[0]=0;
               out_flag[1]=1;
               out_flag[2]=1;
               out_flag[3]=0;
               out_flag[4]=0;
               out_flag[5]=1;
        } /* else */

    printf("\n\nWhat time index of the input file should be the first\n");
    printf("element of the output file?\n");
    scanf("%i",&out1);
    printf("\n\n");

    printf("How many elements should the output file contain?\n");
    scanf("%i",&outN);
    printf("\n\n");

        if (Default != 'n' && Default != 'N')
        {
               printf("Should the time base be normalized to begin at t=1?\n");
               scanf("%s",c);
               if(c==89 || c==121) out_flag[6]=1;
               else out_flag[6]=0;
               printf("\n\n\n");
        } /* if */

        while (1)
        {
               if (fscanf(input_fp, "%f %f %f %f %f %f \n",
               &Time, &Tilt, &Angle, &Velocity, &Temp, &Depth) != EOF)
               {
                    if((Time >= (float)out1) && (Time < (float)(out1+outN)))
                    {
                           if(out_flag[0] && !out_flag[6])
                    fprintf(out_fp,"%f", Time);
                           else if (out_flag[0] && out_flag[6])
                                  fprintf(out_fp, "%f", (float)(i+1));
                           if (out_flag[1]) fprintf(out_fp, "   %f",  Tilt);
                           if (out_flag[2]) fprintf(out_fp, "   %f", Angle);
                           if (out_flag[3]) fprintf(out_fp, "   %f", Velocity);
                           if (out_flag[4]) fprintf(out_fp, "   %f", Temp);
                           if (out_flag[5]) fprintf(out_fp, "   %f", Depth);
                           fprintf(out_fp, "\n");
                           i++ ;
                    } /* if */
                    else if(Time >= (float)(out1+outN))
                    {
                           fclose(input_fp);
                           fclose(out_fp);
                           exit(0);
                    } /* else if */
               } /* if */
               else
               {
                    fclose(input_fp);
                    fclose(out_fp);
                    ExitOnError("EOF encountered in input data file");
               } /* else */
        } /* while */
        exit(1);
}

#if (VERSION==ANSI)
void ExitOnError(char error_txt[])
#elif (VERSION==UNIX)
ExitOnError(error_txt)
```

```
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   exit(1);
}
```

E. ARRAYTEST.C

This program performs various statistical tests on each element of the acoustic array. It is useful for isolating hydrophones which have failed during the experiment. The statistical tests are an adaptation of those found in Reference 11.

```
/****
*  PROGRAM: ARRAYTST vsn 1.0
*  WRITTEN BY: Steven Crocker
*  LAST UPDATE: October 21, 1991
*
*
****/
#define UNIX VERSION            /* either ANSI or UNIX                */
#define CHANNELS 32             /* number of hydrophones on array     */
#define F_SAMPLE 228            /* sampling frequency                 */
#define BUFFER_TIME 60          /* duration of time averages          */

#include<stdio.h>
#include<malloc.h>
#include<math.h>

#if defined ( ANSI )
#include<float.h>
#include<stdlib.h>
int getInput(void);
int moments(float *data, int n, float *ave, float *ave2, float *adev,
       float *sdev, float *svar, float *skew, float *curt)
float *vector(int length);
float **matrix(int row, int col);
int free_matrix(float **m, int row);
void ExitOnError(char error_txt[]);
#elif defined ( UNIX )
int   vector(),
      matrix(),
      getInput(),
            moments(),
      free_matrix(),
      ExitOnError();
#endif

   /* Global Variable Declarations */
float **inSOUND;
int firstBuffer=1;

main()
{
   int i, j=1, k, flag[8];
```

```
float ave, ave2, adev, sdev, svar, skew, curt, *temp;
char fileName[80], key;
FILE *fp1, *fp2, *fp3, *fp4, *fp5, *fp6, *fp7;

inSOUND=(float **)matrix(CHANNELS, BUFFER_TIME*F_SAMPLE);
temp=(float *)vector(BUFFER_TIME*F_SAMPLE);

for (i=0;i<=7;i++) flag[i]=0;
printf("Answer y or n to the following output options.\n");
printf("Average value ( over %i seconds ): ", BUFFER_TIME);
scanf("%s", &key);
if(key==89 || key==121)
{
   flag[1]=1;
   printf("Enter file name: ");
   scanf("%s", fileName);
   printf("\n\n");
   if ((fp1=fopen(fileName, "wt")) == NULL)
      ExitOnError("Error opening file.");
} /* if */

printf("Mean squared value ( over %i seconds ): ", BUFFER_TIME);
scanf("%s", &key);
if(key==89 || key==121)
{
   flag[2]=1;
   printf("Enter file name: ");
   scanf("%s", fileName);
   printf("\n\n");
   if ((fp2=fopen(fileName, "wt")) == NULL)
      ExitOnError("Error opening file.");
} /* if */

printf("Average deviation ( over %i seconds ): ", BUFFER_TIME);
scanf("%s", &key);
if(key==89 || key==121)
{
   flag[3]=1;
   printf("Enter file name: ");
   scanf("%s", fileName);
   printf("\n\n");
   if ((fp3=fopen(fileName, "wt")) == NULL)
      ExitOnError("Error opening file.");
} /* if */

printf("Standard deviation ( over %i seconds ): ", BUFFER_TIME);
scanf("%s", &key);
if(key==89 || key==121)
{
   flag[4]=1;
   printf("Enter file name: ");
   scanf("%s", fileName);
   printf("\n\n");
   if ((fp4=fopen(fileName, "wt")) == NULL)
      ExitOnError("Error opening file.");
} /* if */

printf("Variance ( over %i seconds ): ", BUFFER_TIME);
scanf("%s", &key);
if(key==89 || key==121)
{
   flag[5]=1;
   printf("Enter file name: ");
   scanf("%s", fileName);
   printf("\n\n");
   if ((fp5=fopen(fileName, "wt")) == NULL)
      ExitOnError("Error opening file.");
} /* if */
```

```
  printf("Skewness ( over %i seconds ): ", BUFFER_TIME);
  scanf("%s", &key);
  if(key==89 || key==121)
  {
     flag[6]=1;
     printf("Enter file name: ");
     scanf("%s", fileName);
     printf("\n\n");
     if ((fp6=fopen(fileName, "wt")) == NULL)
        ExitOnError("Error opening file.");
  } /* if */


  printf("Kurtosis ( over %i seconds ): ", BUFFER_TIME);
  scanf("%s", &key);
  if(key==89 || key==121)
  {
     flag[7]=1;
     printf("Enter file name: ");
     scanf("%s", fileName);
     printf("\n\n");
     if ((fp7=fopen(fileName, "wt")) == NULL)
        ExitOnError("Error opening file.");
  } /* if */

  while (j<120)
  {
     getInput();
     if (flag[1]) fprintf(fp1,"%i",j);
     if (flag[2]) fprintf(fp2,"%i",j);
     if (flag[3]) fprintf(fp3,"%i",j);
     if (flag[4]) fprintf(fp4,"%i",j);
     if (flag[5]) fprintf(fp5,"%i",j);
     if (flag[6]) fprintf(fp6,"%i",j);
     if (flag[7]) fprintf(fp7,"%i",j);

     for (i=1;i<=CHANNELS;i++)
     {
        for(k=1;k<=BUFFER_TIME*F_SAMPLE;k++)
           temp[k]=inSOUND[i][k];

        moments(temp,BUFFER_TIME*F_SAMPLE,&ave,&ave2,
                &adev,&sdev,&svar,&skew,&curt);
        if (flag[1]) fprintf(fp1, " %f ", ave);
        if (flag[2]) fprintf(fp2, " %f ", ave2);
        if (flag[3]) fprintf(fp3, " %f ", adev);
        if (flag[4]) fprintf(fp4, " %f ", sdev);
        if (flag[5]) fprintf(fp5, " %f ", svar);
        if (flag[6]) fprintf(fp6, " %f ", skew);
        if (flag[7]) fprintf(fp7, " %f ", curt);
     } /* for */

     if (flag[1]) fprintf(fp1,"\n");
     if (flag[2]) fprintf(fp2,"\n");
     if (flag[3]) fprintf(fp3,"\n");
     if (flag[4]) fprintf(fp4,"\n");
     if (flag[5]) fprintf(fp5,"\n");
     if (flag[6]) fprintf(fp6,"\n");
     if (flag[7]) fprintf(fp7,"\n");
     firstBuffer=0;
     j++;
   } /* while */
   exit(0);
}/*********************************************************************/
 /************************** END main ******************************/
 /*********************************************************************/
/*****
* FUNCTION: getInput()
```

```
*
*  This function handles all acoustic input.  It also provides one of
*  two normal process terminations available in the program.
*
*  Arguments:                       none
*
*  Return value:                    0
*
*  Functions called:                vector()          ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*                                   F_SAMPLE          CHANNELS
*                                   BUFFER_TIME
*
*  Global variables called:         inSOUND[][]        firstBuffer
*
*  Significant memory allocation:   diskBuffer[]
*
*****/
#if defined ( ANSI )
int getInput(void)
#elif defined ( UNIX )
getInput()
#endif
{
      int i, j, items, buffer;
   float *diskBuffer;
   char fileName[80];
   static FILE *fpStatic;

   if (firstBuffer)
   {
      printf("Enter file name for input acoustic data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fpStatic=fopen(fileName, "rb")) == NULL)
         ExitOnError("Error opening INPUT ACOUSTIC data file.");
   } /* if */

   /* Memory Allocation */
       buffer=BUFFER_TIME*F_SAMPLE*CHANNELS;
       diskBuffer=(float*)vector(buffer);

   items=fread((char*)(diskBuffer+1), sizeof(float), buffer, fpStatic);
   if (items==buffer) /*continue*/;
   else if(ferror(fpStatic) != 0)
      ExitOnError("Error encountered while reading input acoustic data");
   else if(feof(fpStatic) != 0)
   {
      printf("\n\t**************************************************\n");
      printf("\t    End of File reached: EXECUTION COMPLETE\n");
      printf("\t**************************************************\n");
      fclose(fpStatic);
            exit(0);
   } /* else if */
   else ExitOnError("Unknown error handling acoustic input file.");

   for (i=1; i<=BUFFER_TIME*F_SAMPLE; i++)
   {
      for (j=1; j<=CHANNELS; j++)
                inSOUND[j][i] = diskBuffer[CHANNELS*(i-1)+j];
       } /* for */

   /* Deallocate Memory */
```

```
   free((char*)diskBuffer);
   return( 0 );
} /************************/
  /***** END getInput *****/
  /************************/
/*****
* FUNCTION: moments()
*
*  This function calculates the mean, mean squared value, average
*  deviation, standard deviation, variance, skewness and kurtosis
*  of a data vector.
*
*  Arguments:                    data            n
*                                ave             ave2
*                                adev            sdev
*                                svar            skew
*                                curt
*
*  Return value:                 0
*
*  Functions called:             ExitOnError()
*
*  Definitions called:           ANSI            UNIX
*
*  Global variables called:      none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
int moments(float *data, int n, float *ave, float *ave2, float *adev,
       float *sdev, float *svar, float *skew, float *curt)
#elif defined ( UNIX )
moments(data,n,ave,ave2,adev,sdev,svar,skew,curt)
int n;
float *data, *ave, *ave2, *adev, *sdev, *svar, *skew, *curt;
#endif
{
   int j;
   float s,p;

   if (n<=1) ExitOnError("n must be at least 2 in moment()");
   s=0.0;
   *ave2=0.0;
   for (j=1;j<=n;j++)
   {
      s += data[j];
      *ave2 += data[j]*data[j];
   } /* for */
   *ave=s/n;
   *ave2 /= n;
   *adev=(*svar)=(*skew)=(*curt)=0.0;
   for (j=1;j<=n;j++)
   {
      *adev += fabs(s=data[j]-(*ave));
      *svar += (p=s*s);
      *skew += (p *= s);
      *curt += (p *= s);
   } /* for */
   *adev /= n;
   *svar /= (n-1);
   *sdev=sqrt(*svar);
   if (*svar)
   {
      *skew /= (n*(*svar)*(*sdev));
      *curt=(*curt)/(n*(*svar)*(*svar))-3.0;
   } /* if */
   else ExitOnError("No skew/kurtosis when variance = 0 (in moment())");
```

```
   return( 0 );
} /**********************/
  /***** END moment *****/
  /**********************/
/*****
* FUNCTION: vector()
*
*  This function allocates memory for UNIT OFFSET vectors.
*
*  Arguments:                       length
*
*  Return value:                    *v
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   v[]
*
*****/
#if defined ( ANSI )
float *vector(int length)
#elif defined ( UNIX )
vector(length)
int length;
#endif
{
   float *v;

   if ((v=(float*)malloc((length+1)*sizeof(float)))==NULL)
      ExitOnError("Memory allocation failure in vector().");
#if defined ( ANSI )
   return v;
#elif defined ( UNIX )
   return (long int)v;
#endif
} /**********************/
  /***** END vector *****/
  /**********************/
/*****
* FUNCTION: matrix()
*
*  This function allocates memory for UNIT OFFSET 2-D arrays.
*
*  Arguments:                       row               col
*
*  Return value:                    **m
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   m[][]
*
*****/
#if defined ( ANSI )
float **matrix(int row, int col)
#elif defined ( UNIX )
matrix(row,col)
int row, col;
#endif
{
   int i;
   float **m;
```

```
   if ((m=(float**)malloc((unsigned)(row+1)*sizeof(float*)))==NULL)
      ExitOnError("Allocation failure 1 in matrix()");
   for (i=1; i<=row; i++)
   {
      if ((m[i]=(float*)malloc((unsigned)(col+1)*sizeof(float)))==NULL)
         ExitOnError("Allocation failure 2 in matrix()");
   } /* for */
#if defined ( ANSI )
   return m;
#elif defined ( UNIX )
   return (long int)m;
#endif
} /**********************/
  /***** END matrix *****/
  /**********************/
/*****
* FUNCTION: free_matrix()
*
*  This function deallocates memory from UNIT OFFSET 2-D arrays.
*
*  Arguments:                       m[][]            row
*
*  Return value:                    0
*
*  Functions called:                none
*
*  Definitions called:              ANSI          UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
void free_matrix(float **m, int row)
#elif defined ( UNIX )
free_matrix(m,row)
float **m;
int row;
#endif
{
   int i;

   for(i=row; i>=1; i--)
      free((char*)m[i]);
   free((char*)m);
   return( 0 );
} /***************************/
  /***** END free matrix *****/
  /***************************/
/*****
* FUNCTION: ExitOnError()
*
*  This function performs an abnormal process termination.
*
*  Arguments:                       error_txt[]
*
*  Return value:                    none
*
*  Functions called:                none
*
*  Definitions called:              ANSI             UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   none
*
*****/
```

```
#if defined ( ANSI )
void ExitOnError(char error_txt[])
#elif defined ( UNIX )
ExitOnError(error_txt)
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   exit(0);
} /***************************/
  /***** END ExitOnError *****/
  /***************************/
```

REFERENCES

1. Munk, W. and Wunsch, C., "Ocean Acoustic Tomography: A Scheme for Large Scale Monitoring," *Deep-Sea Research*, v.26A, pp. 123-161, April 1979.

2. Eldred, R.M., *Doppler Processing of Phase Encoded Underwater Acoustic Signals*, MS Thesis, Naval Postgraduate School, Monterey, CA, September 1990.

3. Kinsler, L.E., and others, *Fundamentals of Acoustics*, 3rd ed., pp. 98-105, John Wiley & Sons, 1982.

4. Coppens, A.B., *Normal Mode Theory for Ocean Waveguides*, pp. 2-4, unpublished manuscript for Naval Postgraduate School course entitled Propagation in the Ocean (PH4453), Monterey, CA, 1990.

5. Chiu, C.S., and Ehret, L.L., Department of Oceanography, Naval Post Graduate School, Monterey, CA, personal communication, August 1991.

6. Dudgeon, D.E. and Mersereau, R.M., *Multidimensional Digital Signal Processing*, pp. 304-305, Prentice-Hall, 1984.

7. Polcari, J.J., *Acoustic Mode Coherence in the Arctic Ocean*, Ph.D. Dissertation, Massachusetts Institute of Technology, Cambridge, MA, May 1986.

8. Frogner, G.R., *Monitoring of Global Acoustic Transmissions: Signal Processing and Preliminary Data Analysis*, MS Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

9. Ort, C.M., *Spatial and Temporal Variability of Cross-Basin Acoustic Ray Paths*, MS Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

10. Ziomek, L.J., *Underwater Acoustics: A Linear Systems Theory Approach*, pp. 120-121, Academic Press, Inc., 1985.

11. Press, W.H. and others., *Numerical Recipes in C*, pp. 85-407, Cambridge University Press, 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6154	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Prof. J.H. Miller, Code EC/Mr Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943	1
4.	Prof. C.S. Chiu, Code OC/Ci Department of Oceanography Naval Postgraduate School Monterey, CA 93943	1
5.	Dr. K. Metzger, Jr. Communications and Signal Processing Laboratory Department of Electrical Engineering and Computer Science North Campus University of Michigan Ann Arbor, MI 48109-2122	1
6.	Prof. T. Birdsall Communications and Signal Processing Laboratory Department of Electrical Engineering and Computer Science University of Michigan Ann Arbor, MI 48109-2122	1
7.	Dr. R.C. Spindel Director, Applied Physics Laboratory University of Washington 1013 Northeast 40th Street Seattle, WA 98105	1
8.	Mr. K. Von der Heydt Woods Hole Oceanographic Institution Woods Hole, MA 02543	1

9. Prof. A.B. Baggeroer 1
Department of Ocean Engineering
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

10. Dr. P. Mikhalevsky 1
SAIC
1710 Goodrich Drive
McLean, VA 22102

11. Dr. W.H. Munk 1
Institute of Geophysics and Planetary Physics
A-025
Scripps Institute of Oceanography
University of California, San Diego
La Jolla, CA 92093

12. Dr. K. Lashkari 1
Monterey Bay Aquarium Research Institute
160 Central Avenue
Pacific Grove, CA 93950

13. LT S.E. Crocker, USN 1
22 Palmer Street
Weymouth, MA 02190

Thesis
C8738 Crocker
c.1 Time domain modal beam-
forming for a near ver-
tical acoustic array.